metran

Release 0.4.0

W.L. Berendrecht

Feb 20, 2024

CONTENTS:

Metran is a package for performing timeseries analysis on multiple timeseries using dynamic factor models.

When modeling multiple groundwater time series within the same hydrological system, it often appears that these components show distinct correlations between locations. Usually large part of the correlation is caused by common input stresses like precipitation and evapotranspiration, which shows up within the deterministic components of the models.

The residual components of the univariate TFN models are often correlated as well. This means that there is spatial correlation which has not been captured by the deterministic component, e.g. because of errors in common input data or due to simplification of the hydrological model leading to misspecification of the deterministic component. We can exploit these correlations by modeling the series simultaneously with a dynamic factor model. Dynamic factor modeling (DFM) is a multivariate timeseries analysis technique used to describe the variation among many variables in terms of a few underlying but unobserved variables called factors.

CHAPTER

ONE

GETTING STARTED

This page explains how to get started with Metran.

1.1 Installing Metran

To install Metran, a working version of Python 3.7 or 3.8 has to be installed on your computer. We recommend using the Anaconda Distribution with Python 3.7 as it includes most of the python package dependencies and the Jupyter Notebook software to run the notebooks. However, you are free to install any Python distribution you want.

To install metran, use:

```
pip install metran
```

To install in development mode, clone the repository, then type the following from the module root directory:

```
pip install -e .
```

1.2 Basic usage

To use Metran, import the metran package:

```
import metran
```

Create Metran model by passing a list of timeseries (measured heads, or timeseries of the residuals of e.g. Pastas timeseries models).

```
list_of_series  # this is a list of series, i.e. [series1, series2, ...]

mt = metran.Metran(list_of_series)
```

To solve the model and determine the specific an dynamic factors:

```
mt.solve()
```

Plotting a simulation for one of the the timeseries:

```
ax = mt.plots.simulation("series1")
```

CHAPTER

TWO

CONCEPTS

The following notebook explains the basic concepts of multivariate timeseries analysis using a technique called dynamic factor modelling as implemented in Metran.

2.1 The Dynamic Factor Model

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import metran

metran.show_versions()
```

```
Python version: 3.10.12 | packaged by conda-forge | (main, Jun 23 2023, 22:40:32) [GCC␣
↪12.3.0]
numpy version: 1.26.4
scipy version: 1.12.0
pandas version: 2.0.3
matplotlib version: 3.8.3
pastas version: 1.3.0
numba version: 0.59.0
lmfit version: 1.2.2
```

Tip: To run this notebook and the related metran model, it is strongly recommended to install Numba (http://numba.pydata.org). This Just-In-Time (JIT) compiler compiles the computationally intensive part of metran model.

When modeling multiple groundwater time series within the same hydrological system, it often appears that these components show distinct correlations between locations. Usually large part of the correlation is caused by common input stresses like precipitation and evapotranspiration, which shows up within the deterministic components of the models.

The residual components of the univariate TFN models are often correlated as well. This means that there is spatial correlation which has not been captured by the deterministic component, e.g. because of errors in common input data or due to simplification of the hydrological model leading to misspecification of the deterministic component. We can exploit these correlations by modeling the series simultaneously with a dynamic factor model. Dynamic factor modeling (DFM) is a multivariate timeseries analysis technique used to describe the variation among many variables in terms of a few underlying but unobserved variables called factors.

This notebook explains the Dynamic Factor Model (DFM) as presented in *Berendrecht and Van Geer, 2016*. It describes the model, model parameters and how the results may be interpreted.

2.1.1 1. Basic multivariate AR(1) model

A general univariate AR(1) model can be written as:

$$x_t = \phi x_{t-1} + \eta_t \tag{2.1}$$

$$n_t = x_t + \varepsilon_t \tag{2.2}$$

with ϕ the AR(1) parameter, η_t a zero mean white noise process, and ε_t the measurement noise. In the current version of `metran` the measurement noise is assumed to be zero, so that $n_t = x_t$.

The multivariate extension of this model is:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix}_t$$

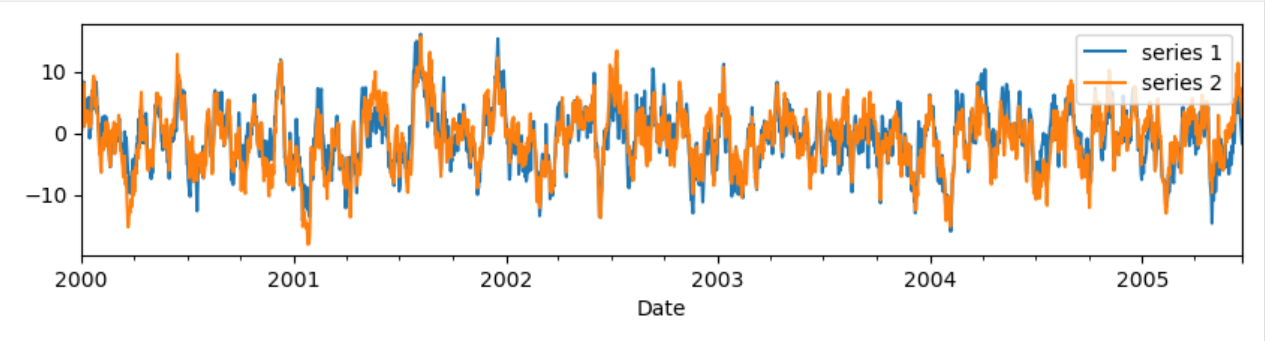
Or:

$$\mathbf{x}_t = \mathbf{\Phi} \mathbf{x}_{t-1} + \eta_t$$

2.1.2 2. Generate synthetic correlated time series

Let us generate time series based on the 2-dimensional model given above. We use the AR(1) model to generate three time series with the AR(1) parameter ϕ: two series as the specific dynamic factor and one series as the common dynamic factor. Combining the specific and common dynamic factors results in two time series which are mutually correlated.

[2]:
```
# seed numpy.random
np.random.seed(20210505)

# define mean and scale (standard deviation for noise series)
mean = np.zeros(3)
scale = [1, 0.6, 2]

# generate noise series that are mutually uncorrelated
noise = np.random.multivariate_normal(mean, np.diag(np.square(scale)), 2001)

# generate AR(1) processes
phi = np.array([0.80, 0.95, 0.90])
a = np.zeros_like(noise)
for i in range(1, noise.shape[0]):
    a[i] = noise[i] + np.multiply(a[i - 1], phi)

# add AR(1) processes to construct two correlated series
s1 = np.add(a[1:, 0], a[1:, 2])
s2 = np.add(a[1:, 1], a[1:, 2])

s = pd.DataFrame(
    data=np.array([s1, s2]).T,
    index=pd.date_range(start="1-1-2000", periods=2000),
    columns=["series 1", "series 2"],
)

s.plot(figsize=(10, 2), xlabel="Date")
```



We can calculated the mean and standard deviation of the generated series and test the correlation between these series. The correlation must be close to the desired correlation defined above.

[3]:
```
print("Mean:")
print(s.mean())
print("\nStandard deviation:")
print(s.std())
print("\nCorrelation:")
print(s.corr())
```

```
Mean:
series 1   -0.847064
series 2   -0.868526
dtype: float64

Standard deviation:
series 1    4.769178
series 2    4.928326
dtype: float64

Correlation:
          series 1  series 2
series 1  1.000000  0.872126
series 2  0.872126  1.000000
```

2.1.3 3. The Dynamic Factor Model

With the Dynamic Factor Model (DFM) we try to decompose series into latent (unobserved) factors describing common and specific dynamics. For the example above, the common dynamic factor describe the all variation that is found in both series. The remaining part of each series is described by the specific dynamic factor.

Mathematically, this can be written as:

$$\begin{bmatrix} n_{1,t} \\ n_{2,t} \end{bmatrix} = \begin{bmatrix} x_{s,1} \\ x_{s,2} \end{bmatrix}_t + \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} x_{c,t}$$

where γ_1 and γ_2 are the factor loadings for series 1 resp. series 2. These factor loadings describe how the series n_1 and n_2 are related to the common dynamic factor.

The specific dynamic factors x_s and common dynamic factor x_c can be described by an AR(1) model as:

$$\mathbf{x}_{s,t} = \begin{bmatrix} \phi_{s,1} & 0 \\ 0 & \phi_{s,2} \end{bmatrix} \mathbf{x}_{s,t-1} + \begin{bmatrix} \eta_{s,1} \\ \eta_{s,2} \end{bmatrix}_t \tag{2.3}$$

$$x_{c,t} = \phi_c x_{c,t-1} + \eta_{c,t} \tag{2.4}$$

The model can also be written in a single matrix notation as:

$$\mathbf{x}_t = \Phi \mathbf{x}_{t-1} + \eta_t \tag{2.5}$$
$$\mathbf{n}_t = \mathbf{Z}\mathbf{x}_t$$

with the state vector $ x‘=:nbsphinx-math:left[begin{array}{c}x_{s,1}\x_{s,2}\x_{c,1}end{array}right]`$, the transition matrix $\mathbf{\Phi} = \begin{bmatrix} \phi_{s,1} & 0 & 0 \\ 0 & \phi_{s,2} & 0 \\ 0 & 0 & \phi_c \end{bmatrix}$, the transition noise vector $\eta = \begin{bmatrix} \eta_{s,1} \\ \eta_{s,2} \\ \eta_c \end{bmatrix}$, and the observation matrix $\mathbf{Z} = \begin{bmatrix} 1 & 0 & \gamma_1 \\ 0 & 1 & \gamma_2 \end{bmatrix}$.

When analyzing more than two series, multiple common dynamic factors may be used. In that case, the equation for the common dynamic factor also becomes a vector equation.

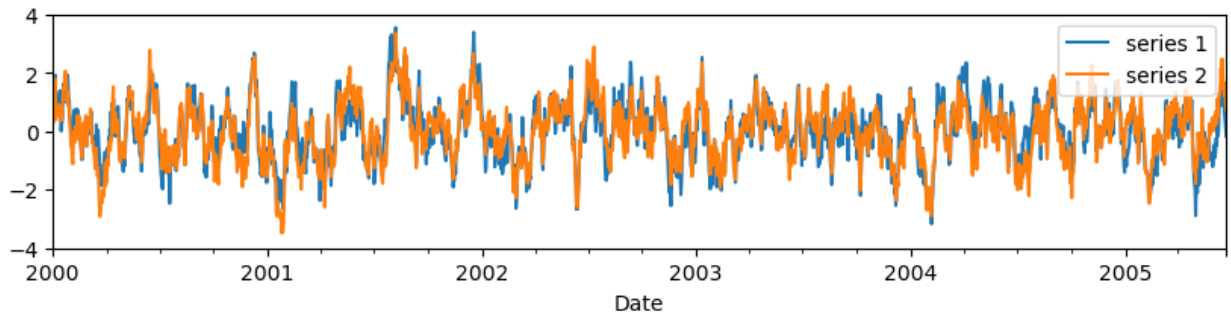
2.1.4 4. Standardization

With the DFM we want to describe the common and specifc dynamics based on the correlation rather than the covariance structure. Therefore, all series are standardized as:

$$\tilde{n}_{i,t} = \frac{n_{i,t} - \mu_{n_i}}{\sigma_{n_i}}$$

This standardization is done internally in `metran`, so there is no need to perform any standardization beforehand. However, as an illustration, the code below shows the standardized series.

[4]:
```
mt = metran.Metran(s)
series_std = mt.standardize(s)
series_std.plot(figsize=(10, 2), xlabel="Date").set_ylim(-4, 4)
```



2.1.5 5. Running the model

Let us now run the model for the generate time series. In this example, we solve the model with `report=False`. This means that no report is shown. Instead, we analyze the results step by step.

[5]:
```
mt = metran.Metran(s)
mt.solve(report=False)
```

```
INFO: Number of factors according to Velicer's MAP test: 1
```

5.1 Factors, communality and specificity

Metran first determines the optimal number of common dynamic factors based on the correlation structure of the time series. For this, the Minimum Average Partial (MAP) test is used (*Velicer, 1976*; *Velicer et al., 2000*). If this test results in 0 factors, then a second test is done based on the Kaiser criterion (*Kaiser, 1960*). In this case, as we can see above, 1 factor has been selected to describe the common dynamics.

Besides, Metran estimates the factor loadings γ_1 and γ_2 using the minimum residual (minres) algorithm (*Harman and Jones, 1966*).

[6]:
```
print("Factors:\n", mt.factors)
```

```
Factors:
 [[0.93540765]
 [0.93540765]]
```

As described in section 3, the factor loadings show the degree to which a factor elaborates a variable (observed series). The sum of squared factor loadings for all common factors for a given series is referred to as the communality. The communality measures the fraction of variance in a given variable explained by all common factors jointly, or in our case, one common factor.

[7]:
```
print("Communality:", mt.get_communality())
```

```
Communality: [0.87498746 0.87498746]
```

The fraction that is unique/specific for each series is referred to as the specificity and is calculated as (1 - communality).

[8]:
```
print("Specificity:", mt.get_specificity())
```

```
Specificity: [0.12501254 0.12501254]
```

5.2 Estimating AR(1) parameters

After the number of factors and associated factor loadings have been estimated, Metran uses an optimization algorithm to estimate the AR(1) model parameters $\phi_{s,1}$, $\phi_{s,2}$, and ϕ_c. Similar to the AR parameter is `pastas`, ϕ is written as:

$$\phi_k = e^{\Delta t_i / \alpha_k}$$

and α_k is being estimated.

As all series have been standardized, the variance of each series is equal to 1. In addtion, we know the communality (and specificity) for each series, which means that we know the variance of the specific and common dynamic factors. As a result, the noise variance parameters of the AR(1) model do not need to be estimated. Instead, Metran calculates them as:

$$q_{s,1} = \left(1 - \phi_{s,1}^2\right) \cdot s_1 \tag{2.7}$$

$$q_{s,2} = \left(1 - \phi_{s,2}^2\right) \cdot s_2 \tag{2.8}$$

$$q_c = \left(1 - \phi_c^2\right) \tag{2.9}$$

with s_1 and s_2 the specificity of series 1 resp. series 2.

The results of the parameter estimation process can be shown using `mt.fit_report()`.

[9]:
```
print(mt.fit_report())
```

```
Fit report Cluster              Fit Statistics
================================================
tmin     None                   obj     2431.34
tmax     None                   nfev         4
freq     D                      AIC     2437.34
solver   ScipySolve

Parameters (3 were optimized)
================================================
                    optimal   stderr initial  vary
series 1_sdf_alpha       10  ±10.00%      10  True
series 2_sdf_alpha       10  ±10.00%      10  True
cdf1_alpha               10  ±10.00%      10  True

Parameter correlations |rho| > 0.5
================================================
None
```

5.3 Metran report

Further output of the Metran model parameters and statistics is given by `mt.metran_report()`. The following results are shown: - nfct: number of factors - fep: percentage of total variance explained by these factors - communality for each series: percentage of variance that a series has in common with other series. - state parameters: - AR(1) parameter ϕ, calculated from the optimized parameter α - variance q of white noise process η - observation parameters: - factor loadings γ for each factor and series - scale: standard deviation σ_n of each series (used for standardization, see section 4) - mean: mean μ_n of each series (used for standardization, see section 4) - state correlations: correlation between specific and/or common dynamic factors

[10]:
```
print(mt.metran_report())
```

```
Metran report Cluster        Factor Analysis
============================================
tmin     None                nfct    1
tmax     None                fep     93.61%
freq     D

Communality
============================================

series 1      87.50%
series 2      87.50%

State parameters
============================================
                   phi         q
series 1_sdf  0.904837  0.022661
series 2_sdf  0.904837  0.022661
cdf1          0.904837  0.181269

Observation parameters
============================================
               gamma1     scale      mean
```

(continues on next page)

(continued from previous page)

```
series 1      0.935408  4.769178 -0.847064
series 2      0.935408  4.928326 -0.868526

State correlations |rho| > 0.5
=============================================
series 1_sdf series 2_sdf -0.88
```

The statistic `fep` is based on the eigenvalues of the correlation matrix. The eigenvalues can be retrieved from the `metran` class.

[11]:
```
mt.eigval
```

[11]:
```
array([1.87212635, 0.12787365])
```

The sum of the eigenvalues always equals the dimension of the correlation matrix, in this case 2.

[12]:
```
round(mt.eigval.sum())
```

[12]:
```
2
```

As we have used 1 eigenvalue (`nfct` = 1), the statistic `fep` is calculated as:

[13]:
```
round(100 * mt.eigval[0] / mt.eigval.sum(), 2)
```

[13]:
```
93.61
```

2.1.6 6. Checking the estimated AR(1) parameters

We can compare the estimate AR(1) parameters ϕ with the AR(1) parameters used to generate the time series.

[14]:
```
print(np.round(np.diagonal(mt.get_transition_matrix()), 2), "vs", phi)
```

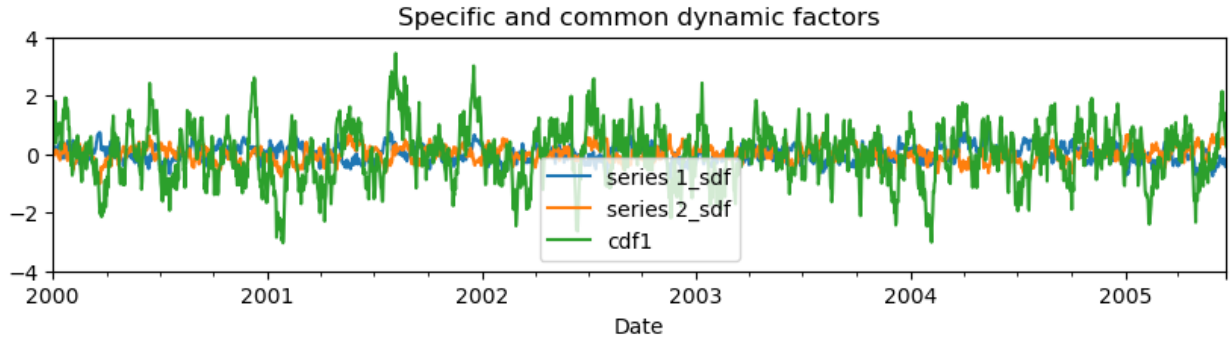
```
[0.9 0.9 0.9] vs [0.8  0.95 0.9 ]
```

The estimated parameters are close to those being used to generate the synthetic series, which means that the model has estimated the autoregression of the latent components well.

2.1.7 7. Decomposition of series

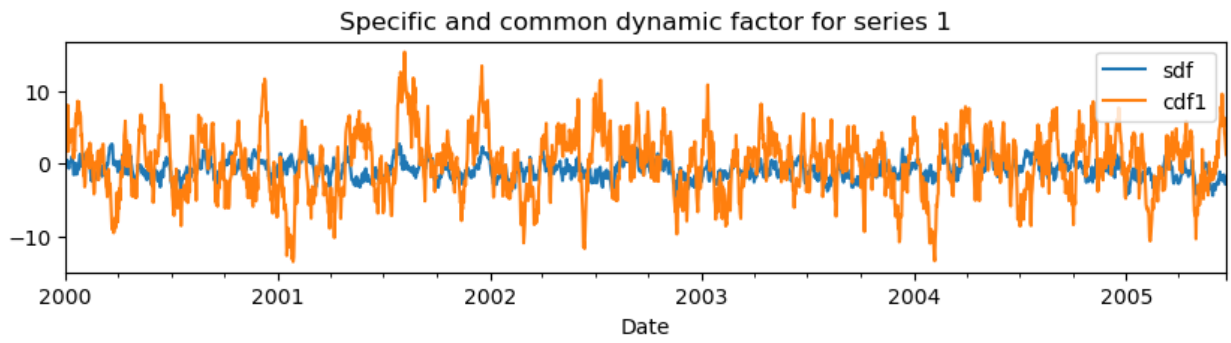
The specific dynamic components (sdf's) $x_{s,1}$ and $x_{s,2}$ can be retrieved from the state vector $\mathbf{x}$.

[15]:
```
mt.get_state_means().plot(
    figsize=(10, 2), xlabel="Date", title="Specific and common dynamic factors"
).set_ylim(-4, 4)
```



Note that the common factor need to be multiplied by the factor loadings, to get the common factor for each series. Furthermore, these results are for the standardized series and need to be rescaled to obtain the unstandardized dynamic factors. Metran has a specific method to obtain the specific and common dynamic factors for each series.

[16]:
```
mt.decompose_simulation(name="series 1").plot(
    figsize=(10, 2),
    xlabel="Date",
    title="Specific and common dynamic factor for series 1",
)
```



We can compare the calculated specificity with the variance of the specific dynamic component divided by the series variance (which is the sum of the specific and common dynamic factor).

[17]:
```
sim1 = mt.decompose_simulation(name="series 1")
sdf1_variance = sim1["sdf"].var() / sim1.sum(axis=1).var()
print("Variance sdf series 1:", "{:.2f}%".format(100 * sdf1_variance))
print("Specificity series 1 :", "{:.2f}%".format(100 * mt.get_specificity()[0]))
```

```
Variance sdf series 1: 6.82%
Specificity series 1 : 12.50%
```

Theoretically, these values must be equal. In practice, they may slightly differ, e.g. due to some correlation between the specific and common dynamic factor. We can test this by calculating the correlation.

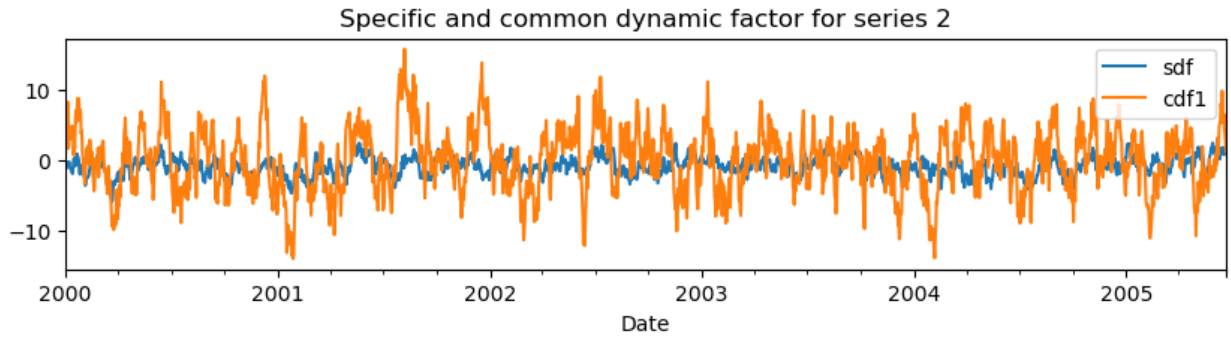
[18]:
```
sim1.corr()
```

[18]:

	sdf	cdf1
sdf	1.00000	0.24815
cdf1	0.24815	1.00000

Similar to series 1, we can decompose series 2 and compare the associated specificity and communality.

[19]:
```
mt.decompose_simulation(name="series 2").plot(
    figsize=(10, 2),
    xlabel="Date",
    title="Specific and common dynamic factor for series 2",
)
```



[20]:
```
sim2 = mt.decompose_simulation(name="series 2")
sdf2_variance = sim2["sdf"].var() / sim2.sum(axis=1).var()
print("Variance sdf series 2:", "{:.2f}%".format(100 * sdf2_variance))
print("Specificity series 2 :", "{:.2f}%".format(100 * mt.get_specificity()[1]))
```

```
Variance sdf series 2: 6.81%
Specificity series 2 : 12.50%
```

[21]:
```
sim2.corr()
```

[21]:

	sdf	cdf1
sdf	1.000000	0.248623
cdf1	0.248623	1.000000

2.1.8 References

- Berendrecht, W.L., F.C. van Geer, 2016. A dynamic factor modeling framework for analyzing multiple groundwater head series simultaneously, Journal of Hydrology, 536, pp. 50-60, DOI.
- Harman, H., Jones, W., 1966. Factor analysis by minimizing residuals (minres). Psychometrika 31, 351–368.
- Kaiser, H.F., 1960. The application of electronic computers to factor analysis. Educ. Psychol. Meas. 20, 141–151.
- Velicer, W.F., 1976. Determining the number of components from the matrix of partial correlations. Psychometrika 41, 321–327.
- Velicer, W.F., Eaton, C.A., Fava, J.L., 2000. Construct explication through factor or component analysis: a review and evaluation of alternative procedures for determining the number of factors or components. In: Goffin, R., Helmes, E. (Eds.), Problems and Solutions in Human Assessment. Springer, US, pp. 41–71.

CHAPTER

THREE

EXAMPLES

The following notebooks contain practical examples showing how Metran works, how to obtain the model results and how to plot them.

The first example shows how to create Metran model and access and visualize the output. The second example shows how Pastas model outputs can be used in Metran.

3.1 Metran practical example

This notebook shows a practical application of Metran on calculated residuals from univariate time series models as published in the article *van Geer and Berendrecht in Stromingen (2015)*.

[1]:
```
import os

import pandas as pd

import metran

metran.show_versions()
```

```
Python version: 3.10.12 | packaged by conda-forge | (main, Jun 23 2023, 22:40:32) [GCC␣
↪12.3.0]
numpy version: 1.26.4
scipy version: 1.12.0
pandas version: 2.2.0
matplotlib version: 3.7.2
pastas version: 1.4.0
numba version: 0.59.0
lmfit version: 1.2.2
```

3.1.1 Read example data

Read residuals from time series analysis models for 5 piezometers at different depths at location B21B0214. (The time series models are not shown here, only the resulting residuals.)

[2]:
```
residuals = {}
rfiles = [
    os.path.join("./data", f) for f in os.listdir("./data") if f.endswith("_res.csv")
]
```

(continues on next page)

(continued from previous page)

```
for fi in rfiles:
    name = fi.split(os.sep)[-1].split(".")[0].split("_")[0]
    ts = pd.read_csv(
        fi, header=0, index_col=0, parse_dates=True, date_format="%Y-%m-%d"
    )
    residuals[name] = ts
```

[3]:
```
# sort names (not necessary, but ensures the order of things)
sorted_names = list(residuals.keys())
sorted_names.sort()
sorted_names
```

[3]:
```
['B21B0214001', 'B21B0214002', 'B21B0214003', 'B21B0214004', 'B21B0214005']
```

3.1.2 Create Metran model

First collect series in a list with their unique IDs.

[4]:
```
series = []

for name in sorted_names:
    ts = residuals[name]
    ts.columns = [name]
    series.append(ts)
```

Create the Metran model and solve.

[5]:
```
mt = metran.Metran(series, name="B21B0214")
mt.solve()
```

```
INFO: Number of factors according to Velicer's MAP test: 1


Fit report B21B0214                  Fit Statistics
=======================================================
tmin     None                        obj        2332.33
tmax     None                        nfev            77
freq     D                           AIC        2344.33
solver   ScipySolve

Parameters (6 were optimized)
=======================================================
                        optimal   stderr initial  vary
B21B0214001_sdf_alpha  5.501017  ±18.98%    10.0  True
B21B0214002_sdf_alpha 13.560042  ±10.04%    10.0  True
B21B0214003_sdf_alpha  4.682870  ±28.86%    10.0  True
B21B0214004_sdf_alpha 11.381674  ±18.22%    10.0  True
B21B0214005_sdf_alpha 13.140605   ±8.48%    10.0  True
cdf1_alpha            22.980925   ±7.43%    10.0  True

Parameter correlations |rho| > 0.5
=======================================================
```

(continues on next page)

(continued from previous page)

```
None

Metran report B21B0214        Factor Analysis
================================================
tmin     None                nfct     1
tmax     None                fep     88.32%
freq     D

Communality
================================================

B21B0214001      73.61%
B21B0214002      87.59%
B21B0214003      93.35%
B21B0214004      91.74%
B21B0214005      81.15%

State parameters
================================================
                     phi         q
B21B0214001_sdf  0.833781  0.080429
B21B0214002_sdf  0.928908  0.017023
B21B0214003_sdf  0.807716  0.023102
B21B0214004_sdf  0.915889  0.013316
B21B0214005_sdf  0.926724  0.026607
cdf1             0.957419  0.083349

Observation parameters
================================================
                   gamma1     scale      mean
B21B0214001      0.857982  5.920523 -0.001924
B21B0214002      0.935874  5.565866 -0.055813
B21B0214003      0.966197  5.702295 -0.001265
B21B0214004      0.957794  5.833851 -0.033373
B21B0214005      0.900857  6.234234 -0.022840

State correlations |rho| > 0.5
================================================
None
```

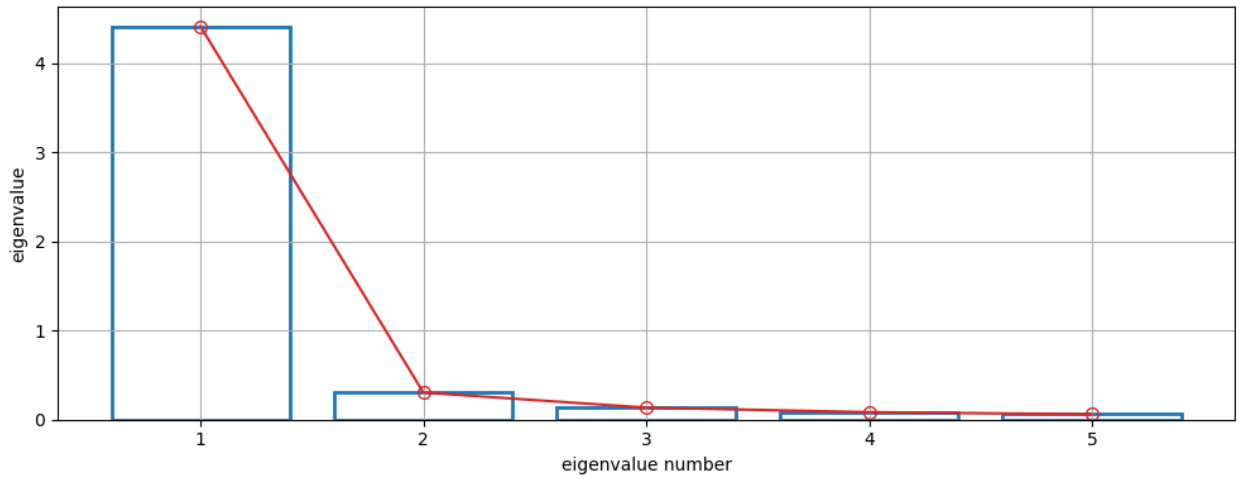
3.1.3 Visualizing and accessing Metran results

The results of the Metran can be visualized using the `Metran.plots` class.

Scree plot

We can draw a scree plot to visualize the eigenvalues (used in determining the number of factors).

[6]:
```
# Plot eigenvalues in scree plot, see e.g. Fig 2 in JoH paper
ax = mt.plots.scree_plot()
```



State means

Plot the calculated state means for each of the specific and common dynamic components:

[7]:
```
mt.get_state_means()
```

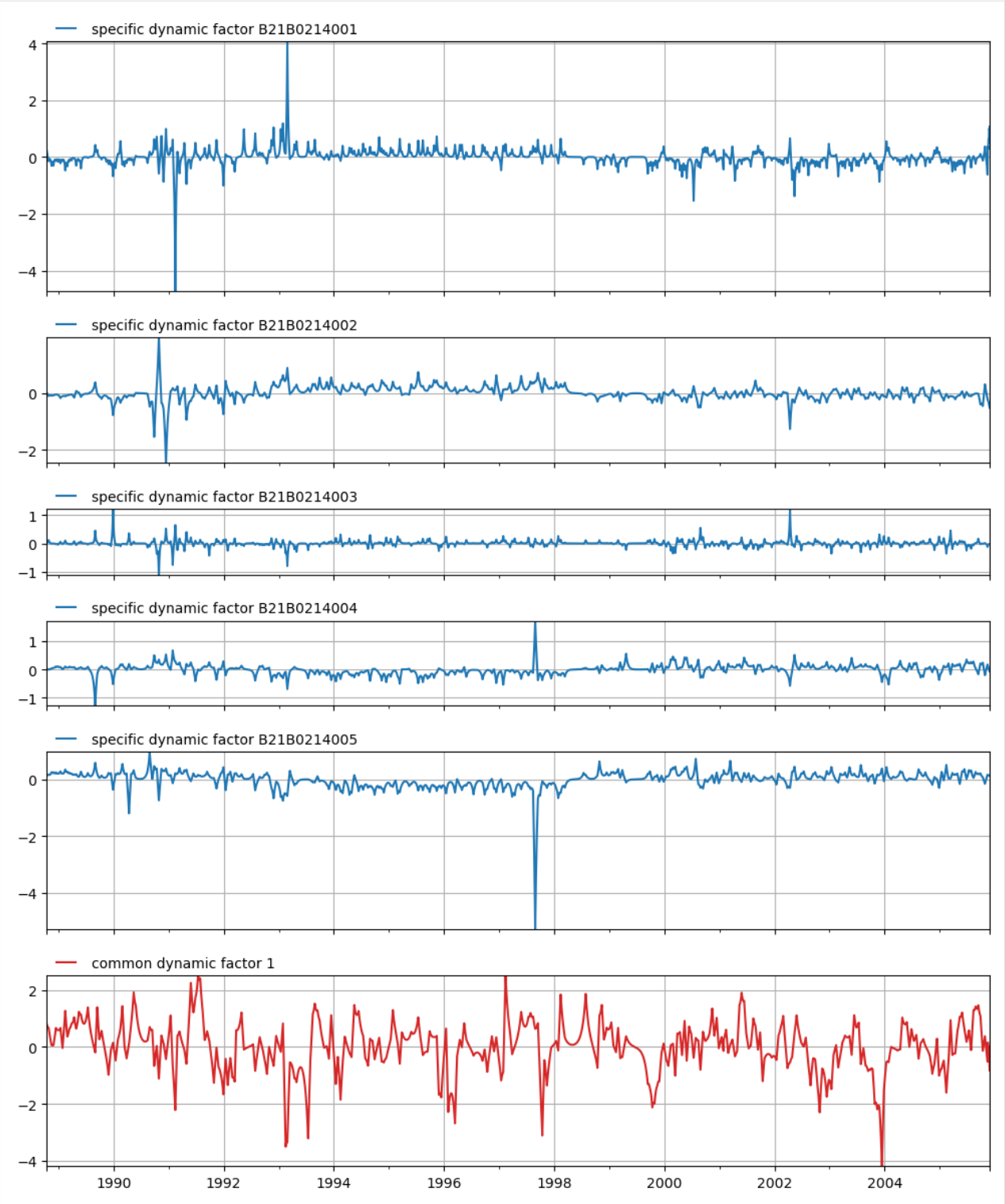
[7]:
```
            B21B0214001_sdf  B21B0214002_sdf  B21B0214003_sdf  \
date
1988-10-14         0.226549         0.021665         0.028548
1988-10-15         0.182900         0.013039         0.026154
1988-10-16         0.145313         0.004483         0.024958
1988-10-17         0.112540        -0.004048         0.024904
1988-10-18         0.083497        -0.012601         0.025990
...                     ...              ...              ...
2005-11-24         0.909463        -0.389068        -0.022535
2005-11-25         0.940340        -0.416086        -0.021679
2005-11-26         1.001189        -0.445368        -0.021816
2005-11-27         0.951887        -0.477072        -0.022951
2005-11-28         1.068061        -0.511373        -0.025137

            B21B0214004_sdf  B21B0214005_sdf      cdf1
date
1988-10-14         0.026005         0.153683  0.809228
1988-10-15         0.022935         0.149910  0.790042
1988-10-16         0.020043         0.147006  0.772353
```

(continues on next page)

(continued from previous page)

```
1988-10-17         0.017306         0.144954  0.756126
1988-10-18         0.014702         0.143742  0.741331
...                     ...              ...       ...
2005-11-24        -0.002155         0.133346 -0.152090
2005-11-25        -0.017868         0.134755 -0.288478
2005-11-26        -0.033719         0.136945 -0.426332
2005-11-27        -0.049831         0.139928 -0.675974
2005-11-28        -0.066328         0.143722 -0.823190

[6255 rows x 6 columns]
```

[8]:
```
axes = mt.plots.state_means(adjust_height=True)
```


specific dynamic factor B21B0214001
specific dynamic factor B21B0214002
specific dynamic factor B21B0214003
specific dynamic factor B21B0214004
specific dynamic factor B21B0214005
common dynamic factor 1
1990
1992
1994
1996
1998
2000
2002
2004

Simulations

The simulated mean values for each time series in our Metran model can be obtained with:

[9]:
```
# Get all (smoothed) simulated state means
means = mt.get_simulated_means()
means.head(10)
```

[9]:

date	B21B0214001	B21B0214002	B21B0214003	B21B0214004	B21B0214005
1988-10-14	5.450000	4.280000	4.620000	4.640000	5.480000
1988-10-15	5.094121	4.132049	4.500647	4.514891	5.348733
1988-10-16	4.781724	3.992287	4.396365	4.399175	5.231284
1988-10-17	4.505266	3.860280	4.306655	4.292536	5.127359
1988-10-18	4.258161	3.735609	4.231336	4.194678	5.036712
1988-10-19	4.034568	3.617869	4.170535	4.105327	4.959142
1988-10-20	3.829204	3.506668	4.124708	4.024229	4.894493
1988-10-21	3.637170	3.401621	4.094661	3.951149	4.842653
1988-10-22	3.453793	3.302355	4.081594	3.885870	4.803556
1988-10-23	3.274477	3.208502	4.087163	3.828191	4.777178

For obtaining the data for a simulation with the observations and an (optional) confidence interval, use `mt.get_simulation()`.

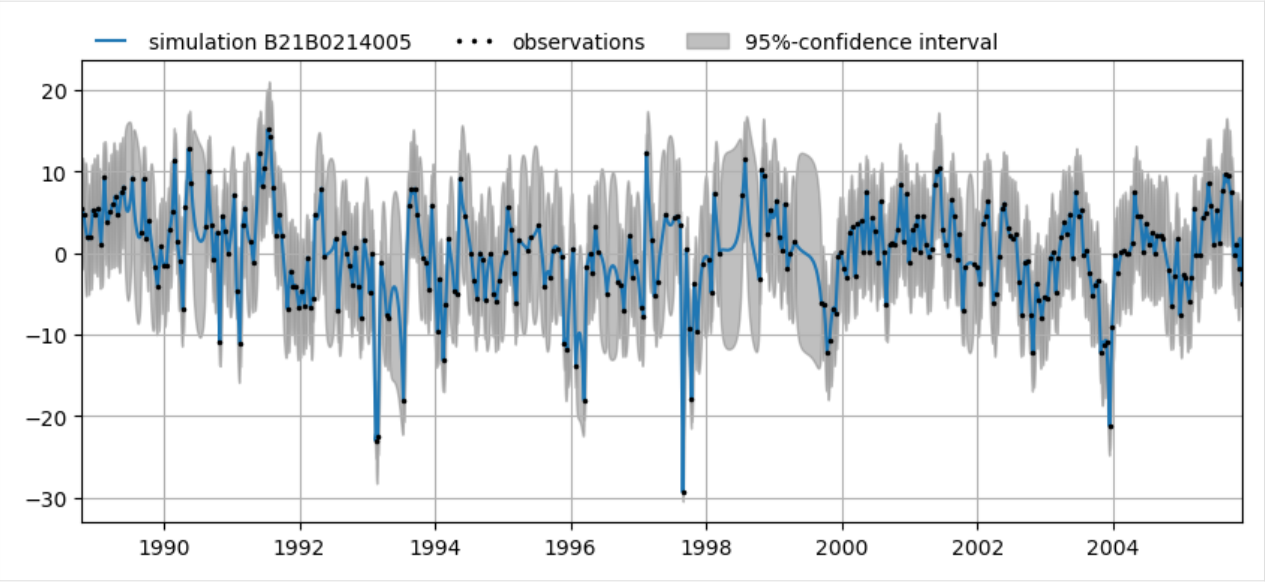
[10]:
```
# Get simulated mean for specific series with/without confidence interval
name = "B21B0214005"
sim = mt.get_simulation(name, alpha=0.05)
sim.head(10)
```

[10]:

date	mean	lower	upper
1988-10-14	5.480000	5.480000	5.480000
1988-10-15	5.348733	1.669956	9.027510
1988-10-16	5.231284	0.273555	10.189012
1988-10-17	5.127359	-0.640237	10.894955
1988-10-18	5.036712	-1.263478	11.336903
1988-10-19	4.959142	-1.669234	11.587519
1988-10-20	4.894493	-1.891043	11.680029
1988-10-21	4.842653	-1.942746	11.628053
1988-10-22	4.803556	-1.824438	11.431550
1988-10-23	4.777178	-1.522468	11.076824

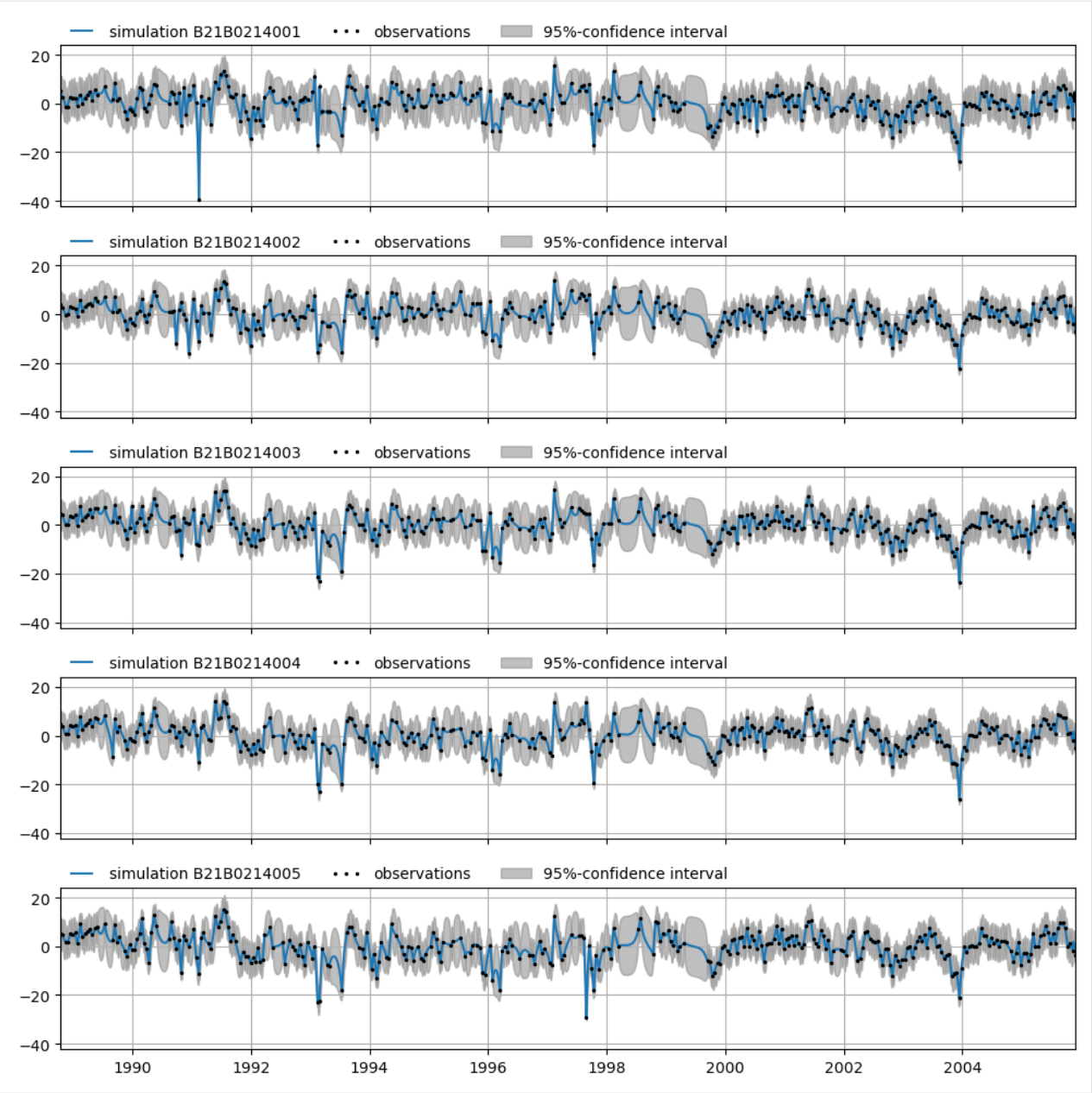
There is also a method to visualize these results for a single time series including optional confidence interval:

[11]:
```
ax = mt.plots.simulation("B21B0214005", alpha=0.05)
```



Or all for all time series:

[12]:
```
axes = mt.plots.simulations(alpha=0.05)
```



Decompositions

The decomposition of a simulation into specific and common dynamic components can be obtained with `mt.decompose_simulation()`.

[13]:
```
# Decomposed simulated mean for specific series
decomposition = mt.decompose_simulation("B21B0214001")
decomposition.head(10)
```

[13]:
```
                 sdf      cdf1
date
1988-10-14  1.339364  4.110636
```

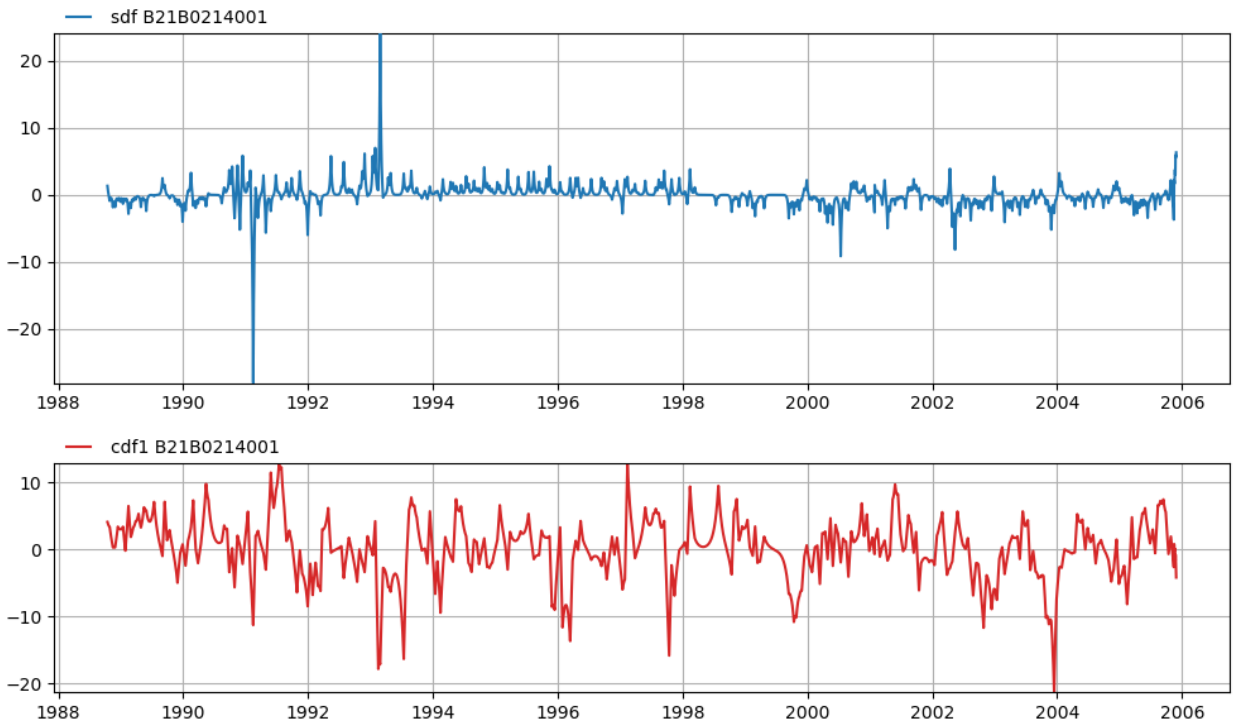
(continues on next page)

(continued from previous page)

```
1988-10-15  1.080942  4.013179
1988-10-16  0.858403  3.923322
1988-10-17  0.664372  3.840894
1988-10-18  0.492420  3.765741
1988-10-19  0.336849  3.697720
1988-10-20  0.192504  3.636701
1988-10-21  0.054601  3.582569
1988-10-22 -0.081428  3.535222
1988-10-23 -0.220092  3.494570
```

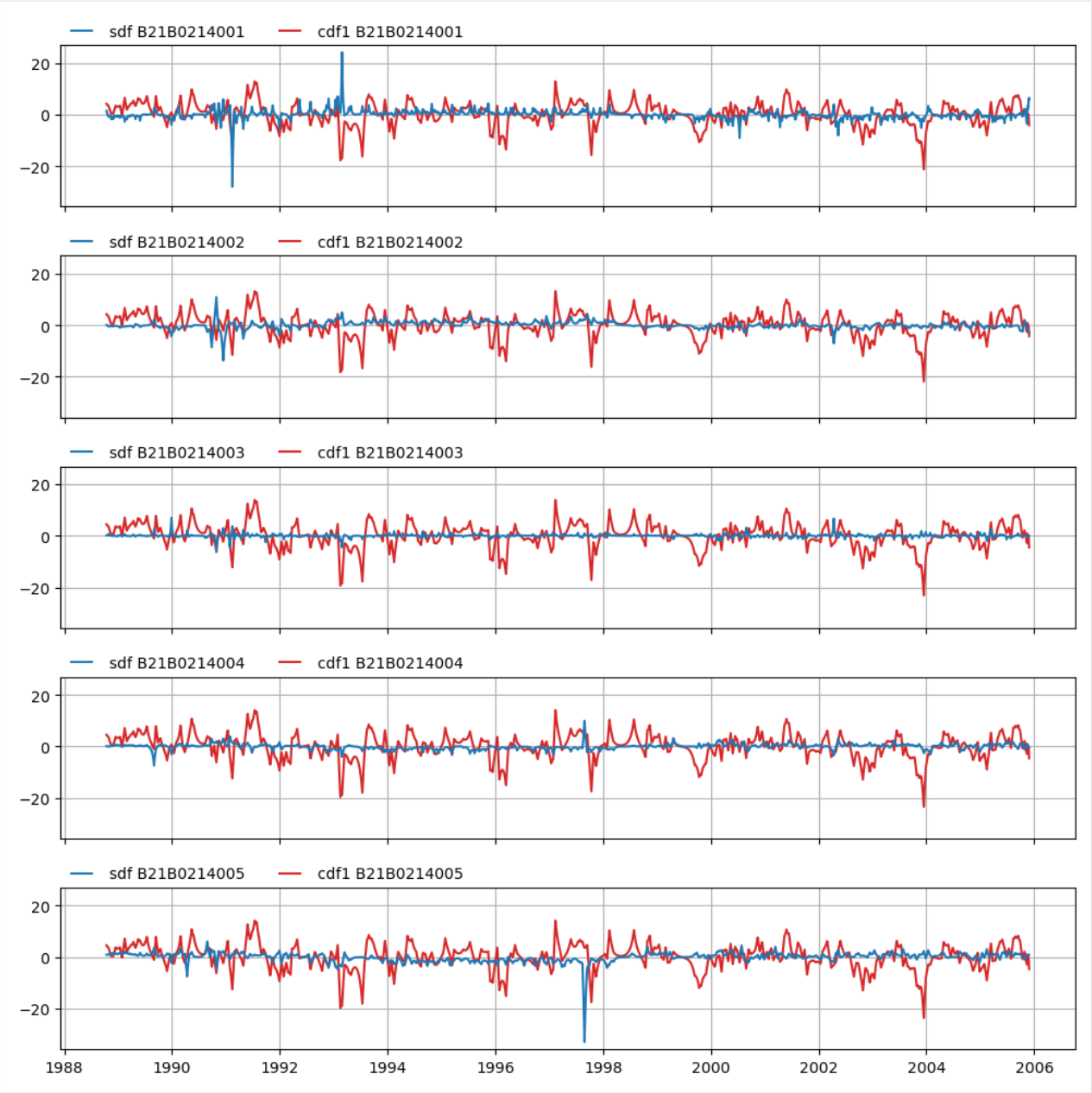
This can also be visualized:

[14]:
```
axes = mt.plots.decomposition("B21B0214001", split=True, adjust_height=True)
```



Or for all time series

[15]:
```
axes = mt.plots.decompositions()
```

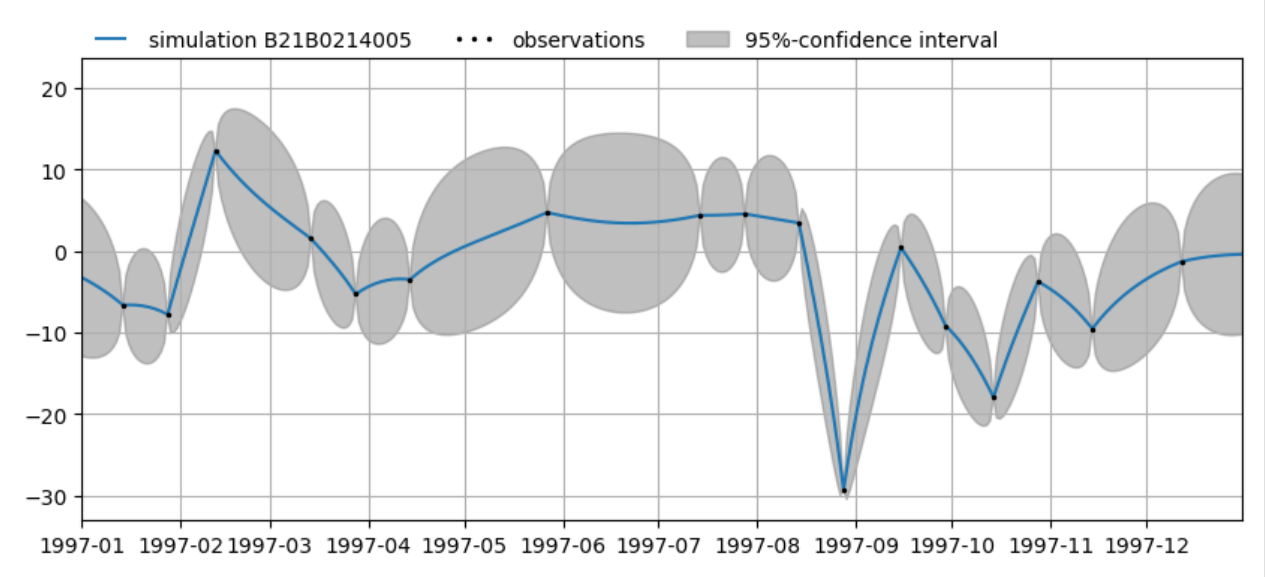


3.1.4 Example application: removing outliers

The Kalman smoother can be re-run after removing (masking) outliers from the observations. This is illustrated below.

First plot the simulation for the original data:

[16]:
```
name = "B21B0214005"
alpha = 0.05

ax1 = mt.plots.simulation(name, alpha=alpha, tmin="1997-01-01", tmax="1997-12-31")
```



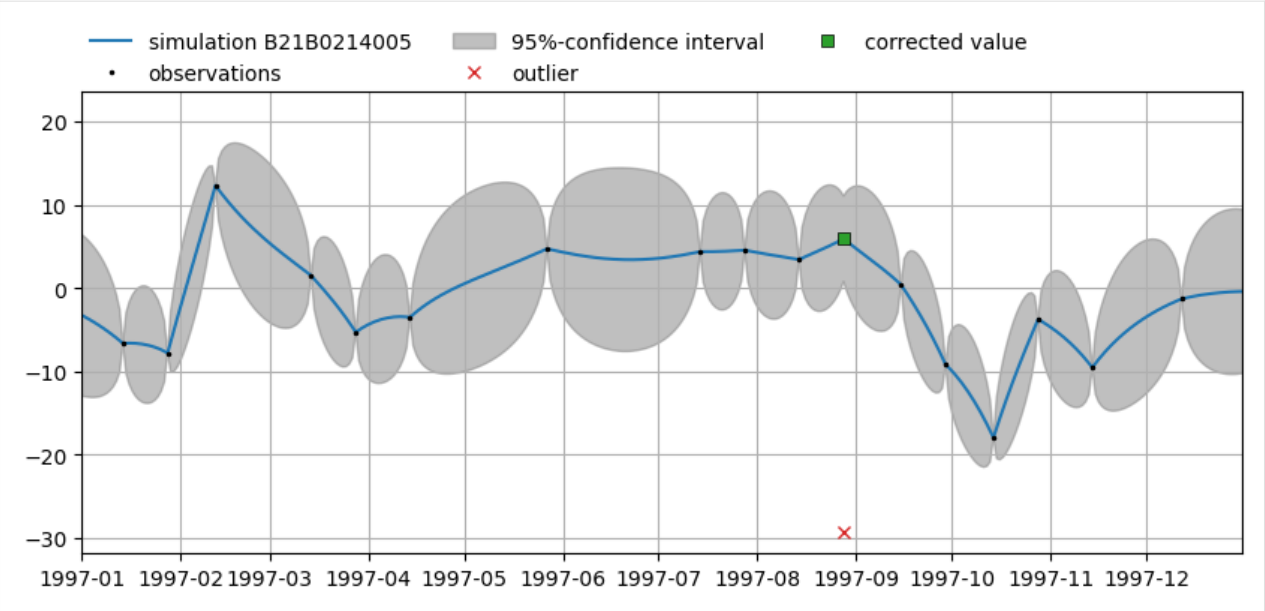
Mask (remove) the outlier on 28 august 1997.

[17]:
```
oseries = mt.get_observations()
mask = (0 * oseries).astype(bool)
mask.loc["1997-8-28", name] = True
mt.mask_observations(mask)
```

Now plot the simulation again. Note the estimated value and its 95%-confidence interval for the observation on 28 August 1997 based on the common dynamic factor.

[18]:
```
# remove outlier from series B21B0214005 at 1997-8-28
# and re-run smoother to get estimate of observation
# (Fig 3 in Stromingen without deterministic component)
ax2 = mt.plots.simulation(name, alpha=alpha, tmin="1997-01-01", tmax="1997-12-31")
sim = mt.get_simulation(name, alpha=None).loc[["1997-8-28"]]

# plot outlier and corrected value
outlier = oseries.loc[["1997-8-28"], name]
ax2.plot(outlier.index, outlier, "C3x", label="outlier")
ax2.plot(sim.index, sim, "C2s", label="corrected value", mec="k", mew=0.5)
ax2.legend(loc=(0, 1), numpoints=1, frameon=False, ncol=3)
```

```
INFO: Running Kalman filter with masked observations.
```



To reset the observations (remove all masks):

[19]:
```
# unmask observations to get original observations
mt.unmask_observations()
```

3.1.5 References

- Van Geer, F.C. en W.L. Berendrecht (2015) Meervoudige tijdreeksmodellen en de samenhang in stijghoogtereeksen. Stromingen 23 nummer 3, pp. 25-36.

3.2 Pastas and Metran example

This notebook shows how output from Pastas time series models can be analyzed using Metran.

[1]:
```
import os

import hydropandas as hpd
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pastas as ps

import metran

ps.logger.setLevel("ERROR")

metran.show_versions()
```

```
Python version: 3.10.12 | packaged by conda-forge | (main, Jun 23 2023, 22:40:32) [GCC␣
↪12.3.0]
numpy version: 1.26.4
```

(continues on next page)

(continued from previous page)

```
scipy version: 1.12.0
pandas version: 2.0.3
matplotlib version: 3.8.3
pastas version: 1.4.0
numba version: 0.59.0
lmfit version: 1.2.2
```

3.2.1 Read data

Load the observed heads from piezometers at different depths at location B21B0214. The outliers (values outside of 5σ (std. dev.)) are removed from the time series.

[2]:
```
oc = hpd.read_dino("./data", subdir=".")
```

[3]:
```
oc
```

[3]:
```
                    x         y       filename source   unit monitoring_well  \
name
B21B0214-001  198085.0  518413.0  B21B0214001_1   dino  m NAP        B21B0214
B21B0214-003  198085.0  518413.0  B21B0214003_1   dino  m NAP        B21B0214
B21B0214-002  198085.0  518413.0  B21B0214002_1   dino  m NAP        B21B0214
B21B0214-004  198085.0  518413.0  B21B0214004_1   dino  m NAP        B21B0214
B21B0214-005  198085.0  518413.0  B21B0214005_1   dino  m NAP        B21B0214

              tube_nr  screen_top  screen_bottom  ground_level  tube_top  \
name
B21B0214-001      1.0        -7.3           -9.3         -0.29      0.24
B21B0214-003      3.0       -24.3          -26.3         -0.29      0.25
B21B0214-002      2.0       -15.3          -17.3         -0.29      0.28
B21B0214-004      4.0       -36.3          -38.3         -0.29      0.23
B21B0214-005      5.0       -51.3          -53.3         -0.29      0.21

              metadata_available  \
name
B21B0214-001               True
B21B0214-003               True
B21B0214-002               True
B21B0214-004               True
B21B0214-005               True

                                                        obs
name
B21B0214-001  GroundwaterObs B21B0214-001
-----metadata-----...
B21B0214-003  GroundwaterObs B21B0214-003
-----metadata-----...
B21B0214-002  GroundwaterObs B21B0214-002
-----metadata-----...
B21B0214-004  GroundwaterObs B21B0214-004
-----metadata-----...
B21B0214-005  GroundwaterObs B21B0214-005
```

(continues on next page)

(continued from previous page)

```
-----metadata-----...
```

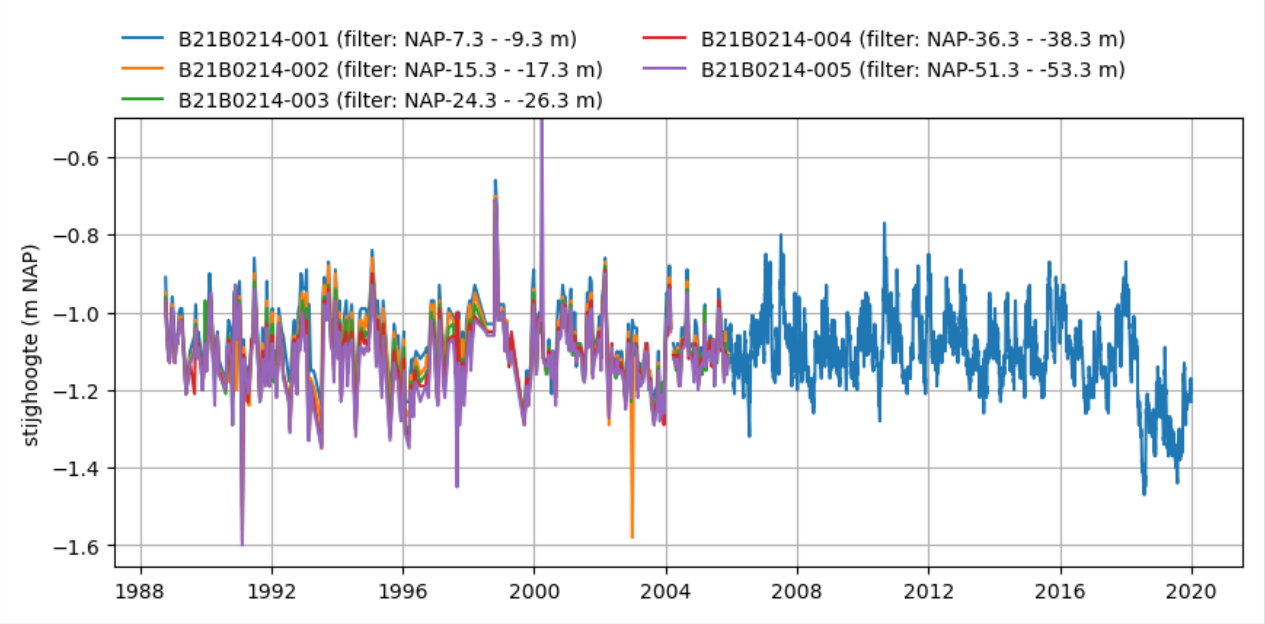
[4]:
```
oseries = {}

for o in oc.obs:
    name = o.name
    o = o["stand_m_tov_nap"].rename(o.name)

    # remove outliers outside 5*std
    mean = o.median()
    std = o.std()
    mask_outliers = (o - mean).abs() > 5 * std

    ts = o.copy()
    ts.loc[mask_outliers] = np.nan

    # store time series
    oseries[name] = ts
```

[5]:
```
# sort the names
sorted_names = list(oseries.keys())
sorted_names.sort()
sorted_names
```

[5]:
```
['B21B0214-001',
 'B21B0214-002',
 'B21B0214-003',
 'B21B0214-004',
 'B21B0214-005']
```

Plot the heads:

[6]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))

for name in sorted_names:
    o = oseries[name]
    ftop = oc.loc[name, "screen_top"]
    fbot = oc.loc[name, "screen_bottom"]
    lbl = f"{name} (filter: NAP{ftop:+.1f} - {fbot:+.1f} m)"
    ax.plot(o.index, o, label=lbl)

ax.set_ylabel("stijghoogte (m NAP)")
ax.legend(loc=(0, 1), ncol=2, frameon=False)
ax.set_ylim(top=-0.5)
ax.grid(True)
```



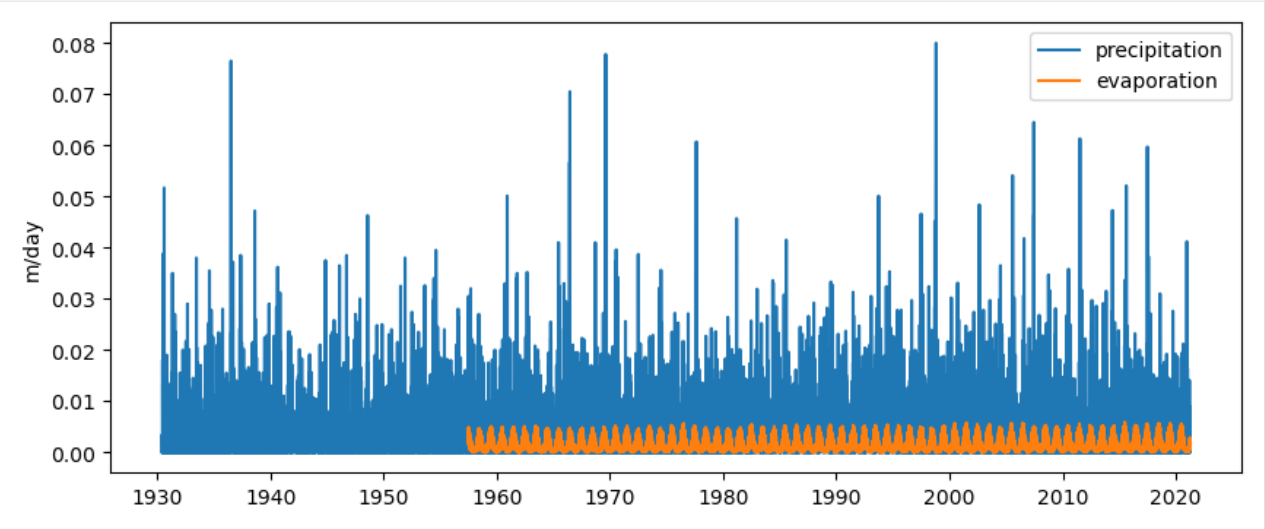
Load the precipitation and evaporation data from two nearby weather stations

[7]:
```
p = pd.read_csv(
    "./data/RD_338.csv", index_col=[0], parse_dates=True, usecols=["YYYYMMDD", "RD"]
)
e = pd.read_csv(
    "./data/EV24_260.csv", index_col=[0], parse_dates=True, usecols=["YYYYMMDD", "EV24"]
)
```

Plot precipitation and evaporation time series

[8]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.plot(p.index, p, label="precipitation")
ax.plot(e.index, e, label="evaporation")
ax.set_ylabel("m/day")
ax.legend(loc="best")
```

[8]:
```
<matplotlib.legend.Legend at 0x7fe032f56770>
```



3.2.2 Build time series models

The time series models attempt to simulate the heads using recharge as stress. The recharge is calculated using $R = P - f \cdot E$, where R is recharge, P is precipitation, E is evaporation and f is factor that is optimized. The model fit results are printed to the console. The model residuals are stored for analysis with Metran.

[9]:
```
# Normalize the index (reset observation time to midnight (the end of the day)).
p.index = p.index.normalize()
e.index = e.index.normalize()

# set tmin/tmax
tmin = "1988-10-14"
tmax = "2005-11-28"

# store models and residuals
models = []
residuals = []

for name in sorted_names:
    # create model
    ml = ps.Model(oseries[name])
    rm = ps.RechargeModel(prec=p, evap=e)
    ml.add_stressmodel(rm)

    # solve model
    ml.solve(tmin=tmin, tmax=tmax, report=False)

    # print fit statistic
    print(name, f"EVP = {ml.stats.evp():.1f}%")

    # store model
    models.append(ml)

    # get residuals
```

(continues on next page)

(continued from previous page)

```
    r = ml.residuals()
    r.name = name
    residuals.append(r)
```

```
B21B0214-001 EVP = 68.4%
B21B0214-002 EVP = 58.9%
B21B0214-003 EVP = 66.8%
B21B0214-004 EVP = 61.3%
B21B0214-005 EVP = 51.0%
```

3.2.3 Build Metran model

A Metran model is created using the residuals of the time series models. By analyzing the model residuals we can determine for example, whether there is a common pattern in the residuals, which could indicate a missing influence, or a shortcoming in the model structure. Additionally we might be able to analyze whether there are still outliers left in our time series.

[10]:
```
mt = metran.Metran(residuals)
mt.solve()
```

```
INFO: Number of factors according to Velicer's MAP test: 1

Fit report Cluster                  Fit Statistics
=======================================================
tmin     None                       obj       3223.12
tmax     None                       nfev            7
freq     D                          AIC       3235.12
solver   ScipySolve

Parameters (6 were optimized)
=======================================================
                        optimal   stderr initial  vary
B21B0214-001_sdf_alpha       10  ±10.00%      10  True
B21B0214-002_sdf_alpha       10  ±10.00%      10  True
B21B0214-003_sdf_alpha       10  ±10.00%      10  True
B21B0214-004_sdf_alpha       10  ±10.00%      10  True
B21B0214-005_sdf_alpha       10  ±10.00%      10  True
cdf1_alpha                   10  ±10.00%      10  True

Parameter correlations |rho| > 0.5
=======================================================
None

Metran report Cluster          Factor Analysis
================================================
tmin     None                nfct      1
tmax     None                fep     81.32%
freq     D

Communality
============================================
```

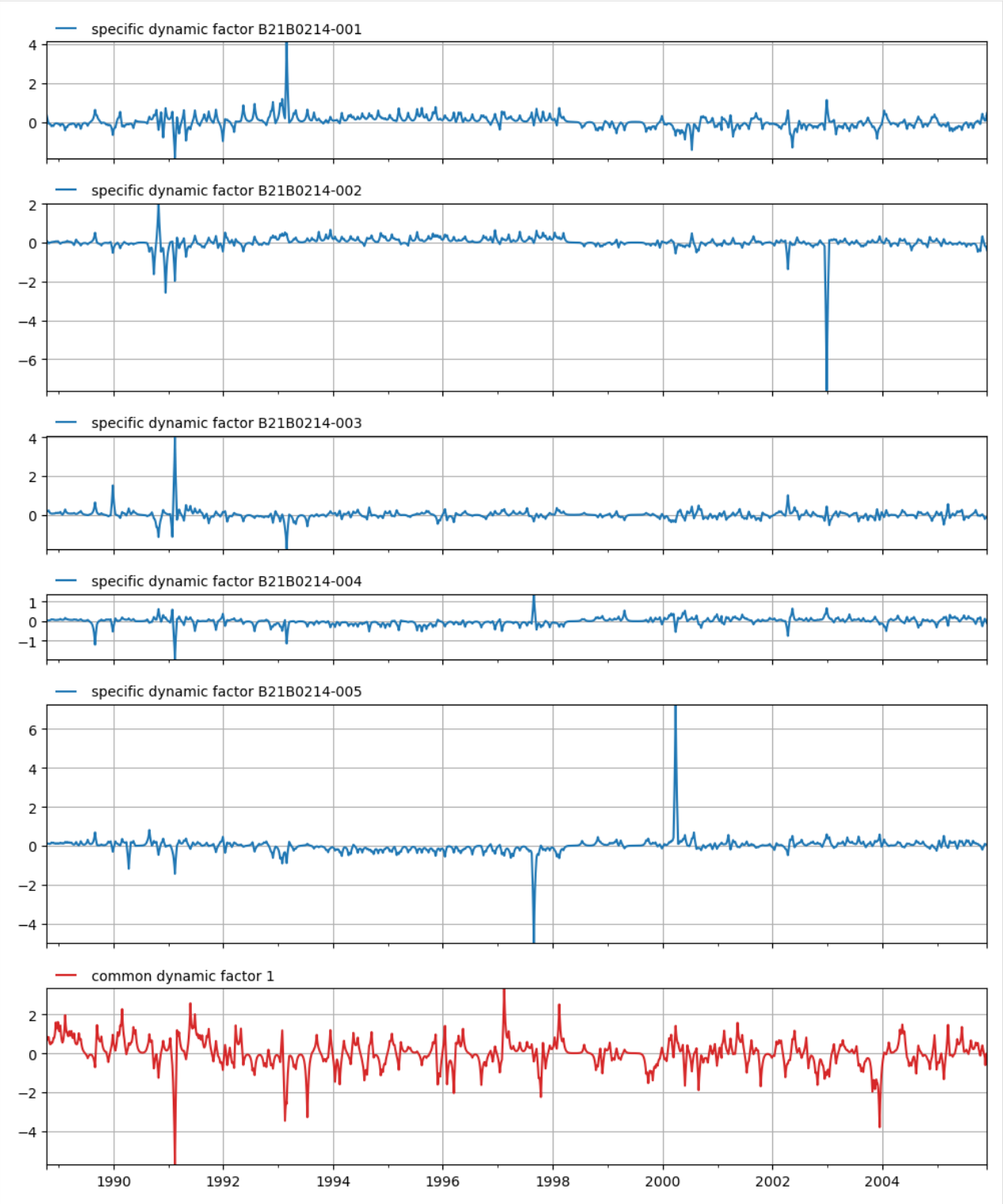
(continues on next page)

(continued from previous page)

```
B21B0214-001      84.36%
B21B0214-002      67.77%
B21B0214-003      88.55%
B21B0214-004      90.71%
B21B0214-005      63.38%

State parameters
================================================
                       phi         q
B21B0214-001_sdf  0.904837  0.028355
B21B0214-002_sdf  0.904837  0.058418
B21B0214-003_sdf  0.904837  0.020764
B21B0214-004_sdf  0.904837  0.016846
B21B0214-005_sdf  0.904837  0.066383
cdf1              0.904837  0.181269

Observation parameters
================================================
                    gamma1     scale      mean
B21B0214-001      0.918464  0.052053 -0.000691
B21B0214-002      0.823243  0.060746 -0.000529
B21B0214-003      0.940984  0.050673 -0.000204
B21B0214-004      0.952399  0.056584  0.000345
B21B0214-005      0.796109  0.070582  0.000577

State correlations |rho| > 0.5
================================================
None
```

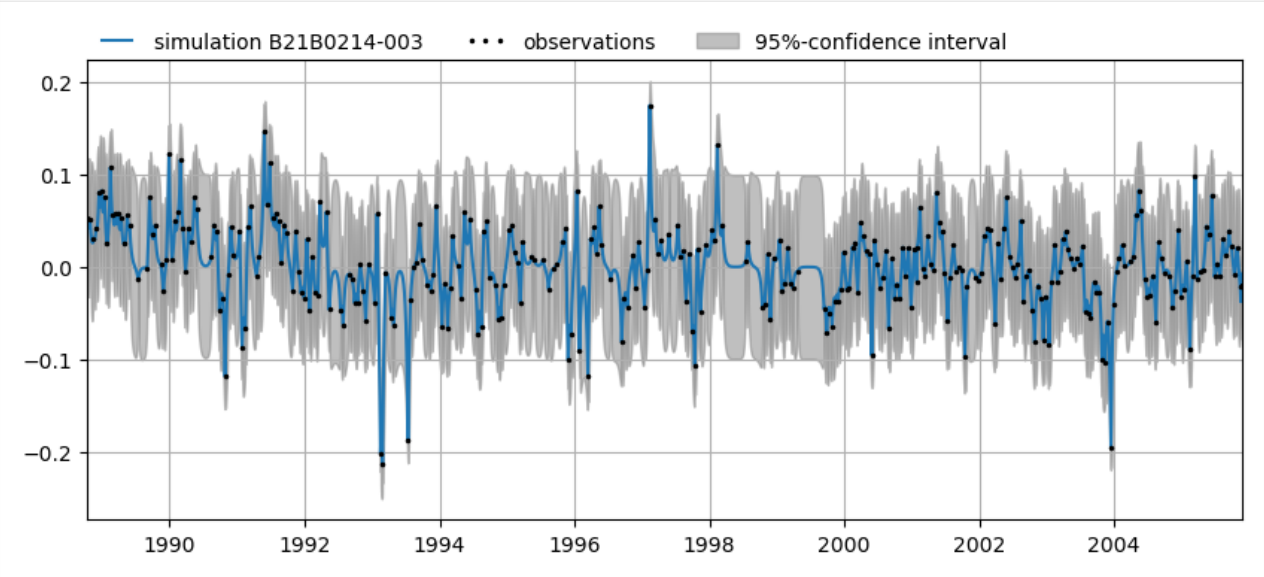
Plot the specific and common dynamic components

[11]:
```
axes = mt.plots.state_means()
```



Plot a simulation, including a confidence interval for B21B0214003.

[12]:
```
ax = mt.plots.simulation(mt.snames[2])
```


simulation B21B0214-003
observations
95%-confidence interval
0.2
0.1
0.0
−0.1
−0.2
1990
1992
1994
1996
1998
2000
2002
2004

CHAPTER

FOUR

API DOCUMENTATION

4.1 Metran

The Metran model class.

class metran.metran.**Metran**(*oseries*, *name='Cluster'*, *freq=None*, *tmin=None*, *tmax=None*)

Class for the Pastas Metran model.

Parameters

- **oseries** (*pandas.DataFrame, list of pandas.Series or pastas.TimeSeries*) – Time series to be analyzed. Index must be DatetimeIndex. The series can be non-equidistant.
- **name** (*str, optional*) – String with the name of the model. The default is 'Cluster'
- **freq** (*str, optional*) – String with the frequency the stressmodels are simulated. Must be one of the following (D, h, m, s, ms, us, ns) or a multiple of that e.g. "7D".
- **tmin** (*str, optional*) – String with a start date for the simulation period (E.g. '1980'). If none is provided, the tmin from the oseries is used.
- **tmax** (*str, optional*) – String with an end date for the simulation period (E.g. '2010'). If none is provided, the tmax from the oseries is used.

Returns

mt – Metran instance.

Return type

metran.Metran

_get_file_info()

Internal method to get the file information.

Returns

file_info – dictionary with file information.

Return type

dict

_get_matrices(*p*, *initial=False*)

Internal method to get all matrices.

Returns all matrices required to define the Metran dynamic factor model.

Parameters

- **p** (*pandas.Series*) – Model parameters.

- **initial** (*bool, optional*) – Determines whether to use initial (True) or optimal (False) parameters. The default is False.

Returns

- *numpy.ndarray* – Transition matrix.
- *numpy.ndarray* – Transition covariance matrix.
- *numpy.ndarray* – Observation matrix.
- *numpy.ndarray* – Observation variance vector.

_init_kalmanfilter(*oseries*, *engine='numba'*)

Internal method, initialize Kalmanfilter for sequential processing.

Parameters

- **oseries** (*pandas.DataFrame*) – Series being processed by the Kalmanfilter.
- **engine** (*str, optional*) – Engine used for the Kalman filter, by default 'numba' which is the fastest choice but 'numpy' is also available, but is slower.

Return type
None.

_phi(*alpha*)

Internal method to calculate autoregressive model parameter.

Autoregressive model parameter is calculated based on parameter alpha.

Parameters
alpha (*float*) – model parameter

Returns
autoregressive model parameter

Return type
float

_run_kalman(*method*, *p=None*)

Internal method to (re)run Kalman filter or smoother.

Parameters

- **method** (*str, optional*) – Use "filter" to run Kalman filter, and "smoother" to run Kalman smoother. The default is "smoother".
- **p** (*pandas.Series*) – Model parameters. The default is None.

Return type
None.

decompose_simulation(*name*, *p=None*, *standardized=False*, *method='smoother'*)

Decompose simulation into specific and common dynamic components.

Method to get for observed series filtered/smoothed estimate decomposed into specific dynamic component (sdf) and the sum of common dynamic components (cdf).

Parameters

- **name** (*str*) – name of series to be obtained.
- **p** (*pandas.Series*) – Model parameters. The default is None.

- **standardized** (*bool, optional*) – If True, obtain estimates for standardized series. If False, obtain estimates for unstandardized series. The default is False.
- **method** (*str, optional*) – Use "filter" to obtain filtered estimates, and "smoother" to obtain smoothed estimates. The default is "smoother".

Returns
df – DataFrame with specific and common dynamic component for series with name 'name'.

Return type
pandas.DataFrame

fit_report(*output='full'*)

Method that reports on the fit after a model is optimized.

Parameters
output (*str, optional*) – If any other value than "full" is provided, the parameter correlations will be removed from the output.

Returns
report – String with the report.

Return type
str

Examples

This method is called by the solve method if report=True, but can also be called on its own:

```
>>> print(mt.fit_report())
```

get_communality()

Get fraction that is explained by the common dynamic factor(s).

Calculate communality for each series.

Returns
For each series the communality, a value between 0 and 1. A value of 0 means that the series has no variation in common with other series. A value of 1 means that the series has all variation in common.

Return type
numpy.ndarray

get_factors(*oseries=None*)

Method to get factor loadings based on factor analysis.

This method also gets some relevant results from the factor analysis including the eigenvalues and percentage explained by factors (fep).

Parameters
oseries (*pandas.DataFrame, optional*) – Series to be analyzed. The default is None.

Returns
factors – Factor loadings as estimated using factor analysis

Return type
numpy.ndarray

`get_mle`(*p*)

Method to obtain maximum likelihood estimate based on Kalman filter.

Parameters
p (*pandas.Series*) – Model parameters.

Returns
mle – Maximum likelihood estimate.

Return type
float

`get_observation_matrix`(*p=None*, *initial=False*)

Method to get observation matrix of the Metran dynamic factor model.

Parameters

- **p** (*pandas.Series, optional*) – Model parameters. The default is None.
- **initial** (*bool, optional*) – Determines whether to use initial (True) or optimal (False) parameters. The default is False.

Returns
observation_matrix – Observation matrix

Return type
numpy.ndarray

`get_observation_variance`()

Method to get observation matrix.

Currently the observation variance is zero by default.

Returns
observation_variance – Observation variance vector

Return type
numpy.ndarray

`get_observations`(*standardized=False*, *masked=False*)

Returns series as available in Metran class.

Parameters

- **standardized** (*bool, optional*) – If True, obtain standardized observations. If False, obtain unstandardized observations. The default is False.
- **masked** (*boolean*) – If True, return masked observations. The default is False.

Returns
Time series.

Return type
pandas.DataFrame

`get_parameters`(*initial=False*)

Method to get all parameters from the individual objects.

Parameters
initial (*bool, optional*) – True to get initial parameters, False to get optimized parameters. If optimized parameters do not exist, return initial parameters.

Returns
parameters – initial or optimal parameters.

Return type
pandas.Series

`get_scaled_observation_matrix`(*p=None*)

Method scale observation matrix by standard deviations of oseries.

Returns

- **observation_matrix** (*numpy.ndarray*) – scaled observation matrix
- **p** (*pandas.Series*) – Model parameters. The default is None.

`get_simulated_means`(*p=None*, *standardized=False*, *method='smoother'*)

Method to calculate simulated means.

Simulated means are the filtered/smoothed mean estimates for the observed series.

Parameters

- **p** (*pandas.Series*) – Model parameters. The default is None.
- **standardized** (*bool, optional*) – If True, obtain estimates for standardized series. If False, obtain estimates for unstandardized series. The default is False.
- **method** (*str, optional*) – Use "filter" to obtain filtered estimates, and "smoother" to obtain smoothed estimates. The default is "smoother".

Returns
simulated_means – Filtered or smoothed estimates for observed series.

Return type
pandas.DataFrame

`get_simulated_variances`(*p=None*, *standardized=False*, *method='smoother'*)

Method to calculate simulated variances,

The simulated variances are the filtered/smoothed variances for the observed series.

Parameters

- **p** (*pandas.Series*) – Model parameters. The default is None.
- **standardized** (*bool, optional*) – If True, obtain estimates for standardized series. If False, obtain estimates for unstandardized series. The default is False.
- **method** (*str, optional*) – Use "filter" to obtain filtered estimates, and "smoother" to obtain smoothed estimates. The default is "smoother".

Returns
simulated_variances – Filtered or smoothed variances for observed series.

Return type
pandas.DataFrame

`get_simulation`(*name*, *p=None*, *alpha=0.05*, *standardized=False*, *method='smoother'*)

Method to calculate simulated means for specific series.

Optionally including 1-alpha confidence interval.

Parameters

- **name** (*str*) – name of series to be obtained
- **p** (*pandas.Series*) – Model parameters. The default is None.

- **alpha** (*float, optional*) – Include (1-alpha) confidence interval in DataFrame. The value of alpha must be between 0 and 1. If None, no confidence interval is returned. The default is 0.05.
- **standardized** (*bool, optional*) – If True, obtain estimates for standardized series. If False, obtain estimates for unstandardized series. The default is False.
- **method** (*str, optional*) – Use "filter" to obtain filtered estimates, and "smoother" to obtain smoothed estimates. The default is "smoother".

Returns
proj – filtered or smoothed estimate (mean) for series 'name', optionally with 'lower' and 'upper' as lower and upper bounds of 95% confidence interval.

Return type
pandas.DataFrame

get_specificity()

Get fraction that is explained by the specific dynamic factor.

Calculate specificity for each series. The specificity is equal to (1 - communality).

Returns
For each series the specificity, a value between 0 and 1. A value of 0 means that the series has all variation in common with other series. A value of 1 means that the series has no variation in common.

Return type
numpy.ndarray

get_state(*i*, *p=None*, *alpha=0.05*, *method='smoother'*)

Get filtered or smoothed mean for specific state.

Optionally including the 1-alpha confidence interval.

Parameters

- **i** (*int*) – index of state vector to be obtained
- **p** (*pandas.Series*) – Model parameters. The default is None.
- **alpha** (*float, optional*) – Include (1-alpha) confidence interval in DataFrame. The value of alpha must be between 0 and 1. If None, no confidence interval is returned. The default is 0.05.
- **method** (*str, optional*) – Use "filter" to obtain filtered variances, and "smoother" to obtain smoothed variances. The default is "smoother".

Returns
state – ith filtered or smoothed state (mean), optionally with 'lower' and 'upper' as lower and upper bounds of 95% confidence interval.

Return type
pandas.DataFrame

get_state_means(*p=None*, *method='smoother'*)

Method to get filtered or smoothed state means.

Parameters

- **p** (*pandas.Series*) – Model parameters. The default is None.
- **method** (*str, optional*) – Use "filter" to obtain filtered states, and "smoother" to obtain smoothed states. The default is "smoother".

Returns
state_means – Filtered or smoothed states. Column names refer to specific dynamic factors (sdf) and common dynamic factors (cdf)

Return type
pandas.DataFrame

`get_state_variances`(*p=None*, *method='smoother'*)

Method to get filtered or smoothed state variances.

Parameters

- **p** (*pandas.Series*) – Model parameters. The default is None.
- **method** (*str, optional*) – Use "filter" to obtain filtered variances, and "smoother" to obtain smoothed variances. The default is "smoother".

Returns
state_variances – Filtered or smoothed variances. Column names refer to specific dynamic factors (sdf) and common dynamic factors (cdf)

Return type
pandas.DataFrame

`get_transition_covariance`(*p=None*, *initial=False*)

Get transition covariance matrix of the Metran dynamic factor model.

Parameters

- **p** (*pandas.Series, optional*) – Model parameters. The default is None.
- **initial** (*bool, optional*) – Determines whether to use initial (True) or optimal (False) parameters. The default is False.

Returns
transition_covariance – Transition covariance matrix

Return type
numpy.ndarray

`get_transition_matrix`(*p=None*, *initial=False*)

Method to get transition matrix of the Metran dynamic factor model.

Parameters

- **p** (*pandas.Series, optional*) – Model parameters. The default is None.
- **initial** (*bool, optional*) – Determines whether to use initial (True) or optimal (False) parameters. The default is False.

Returns
transition_matrix – Transition matrix

Return type
numpy.ndarray

`get_transition_variance`(*p=None*, *initial=False*)

Get the transition variance vector.

The transition variance vector is obtained by extracting the diagonal of the transition covariance matrix.

Parameters

- **p** (*pandas.Series, optional*) – Model parameters. The default is None.

- **initial** (*bool, optional*) – Determines whether to use initial (True) or optimal (False) parameters. The default is False.

Returns
transition_variance – Transition variance vector

Return type
numpy.ndarray

mask_observations(*mask*)

Mask observations for processing with Kalman filter or smoother.

This method does NOT change the oseries itself. It only masks (hides) observations while running the Kalman filter or smoother, so that these observations are not used for updating the Kalman filter/smoother.

Parameters
mask (*pandas.DataFrame*) – DataFrame with shape of oseries containing 0 or False for observation to be kept and 1 or True for observation to be masked (hidden).

Return type
None.

metran_report(*output='full'*)

Method that reports on the metran model results.

Parameters
output (*str, optional*) – If any other value than "full" is provided, the state correlations will be removed from the output.

Returns
report – String with the report.

Return type
str

Examples

This method is called by the solve method if report=True, but can also be called on its own:

```
>>> print(mt.metran_report())
```

property nparam

property nstate

set_init_parameters()

Method to initialize parameters to be optimized.

Return type
None

set_observations(*oseries*)

Rework oseries to pandas.DataFrame for further use in Metran class.

Parameters

- **oseries** (*pandas.DataFrame*) –
- **pandas.Series/pandas.DataFrame/pastas.TimeSeries** (*or list/tuple of*) – Time series to be analyzed.

Raises
Exception –

- if a DataFrame within a list/tuple has more than one column - if input type is not correct - if number of series is less than 2 - if index of Series/DataFrame is not a DatetimeIndex

Return type
None.

solve(*solver=None*, *report=True*, *engine='numba'*, ***kwargs*)

Method to solve the time series model.

Parameters

- **solver** (*metran.solver.BaseSolver class, optional*) – Class used to solve the model. Options are: mt.ScipySolve (default) or mt.LmfitSolve. A class is needed, not an instance of the class!
- **report** (*bool, optional*) – Print reports to the screen after optimization finished. This can also be manually triggered after optimization by calling print(mt.fit_report()) or print(mt.metran_report()) on the Metran instance.
- **engine** (*str, optional*) – Engine used for the Kalman filter, by default 'numba' which is the fastest choice but 'numpy' is also available, but is slower.
- ****kwargs** (*dict, optional*) – All keyword arguments will be passed onto minimization method from the solver.

Notes

- The solver object including some results are stored as mt.fit. From here one can access the covariance (mt.fit.pcov) and correlation matrix (mt.fit.pcor).
- The solver returns a number of results after optimization. These are stored in mt.fit.result and can be accessed from there.

standardize(*oseries*)

Method to standardize series.

Standardized by subtracting mean and dividing by standard deviation.

Parameters
oseries (*pandas.DataFrame*) – series to be standardized

Returns
standardized series

Return type
pandas.DataFrame

test_cross_section(*oseries=None*, *min_pairs=None*)

Method to test whether series have enough cross-sectional data.

Default threshold value is defined by self.settings["min_pairs"].

Parameters

- **oseries** (*pandas.DataFrame, optional*) – Time series to be evaluated. The default is None.
- **min_pairs** (*int, optional*) – Minimum number of cross-sectional data for each series. Should be greater than 1. The default is None.

Raises
Exception – If one of the series has less than min_pairs of cross-sectional data and exception is raised.

Return type
None.

truncate(*oseries*)

Method to set start and end of series.

If tmin and/or tmax have been defined in self.settings, use these dates to trucate series. Dates with only NaN are being removed.

Parameters
oseries (*pandas.DataFrame*) – series to be tructated

Returns
truncated series

Return type
pandas.DataFrame

unmask_observations()

Method to unmask observation and reset observations.

Return type
None

4.2 Factor Analysis

FactorAnalysis class for Metran in Pastas.

class metran.factoranalysis.**FactorAnalysis**(*maxfactors=None*)

Class to perform a factor analysis for the Pastas Metran model.

Parameters
maxfactors (*int, optional.*) – maximum number of factors to select. The default is None.

Examples

A minimal working example of the FactorAnalysis class is shown below:

```
>>> fa = FactorAnalysis()
>>> factors = fa.solve(oseries)
```

get_eigval_weight()

Method to get the relative weight of each eigenvalue.

Returns
All eigenvalues as a fraction of the sum of eigenvalues.

Return type
numpy.ndarray

solve(*oseries*)

Method to perform factor analysis.

Factor analysis is based on the minres algorithm. The number of eigenvalues is determined by MAP test. If more than one eigenvalue is used, the factors are rotated using orthogonal rotation.

Parameters
oseries (*pandas.DataFrame*) – Object containing the time series. The series can be non-equidistant.

Raises
Exception – If no proper factors can be derived from the series.

Returns
factors – Factor loadings.

Return type
numpy.ndarray

4.3 Kalman Filter

This module contains the Kalman filter class for Metran and associated filtering and smoothing methods.

class metran.kalmanfilter.**SPKalmanFilter**(*engine='numba'*)

Kalman filter class for Metran.

Parameters
engine (*str, optional*) – Engine to be used to run sequential Kalman filter. Either "numba" or "numpy". The default is "numba".

Returns
kf – Metran SPKalmanfilter instance.

Return type
kalmanfilter.SPKalmanFilter

decompose(*observation_matrix*, *method='smoother'*)

Method to decompose simulated means.

Decomposition into specific dynamic factors (sdf) and common dynamic factors (cdf).

Parameters

- **observation_matrix** (*numpy.ndarray*) – Observation matrix for projecting states.
- **method** (*str, optional*) – If "filter", use Kalman filter to obtain estimates. If "smoother", use Kalman smoother. The default is "smoother".

Returns

- **sdf_means** (*list*) – List of specific dynamic factors for each time step.
- **cdf_means** (*list*) – List of common dynamic factor(s) for each time step.

get_mle(*warmup=1*)

Method to calculate maximum likelihood estimate.

Parameters
warmup (*int, optional*) – Number of time steps to skip. The default is 1.

Returns
mle – Maximum likelihood estimate.

Return type
float

`init_states()`

Method to initialize state means and covariances.

Return type
None.

`run_filter(initial_state_mean=None, initial_state_covariance=None, engine=None)`

Method to run the Kalman Filter.

This is a sequential processing implementation of the Kalman filter requiring a diagonal observation error covariance matrix. The algorithm allows for missing data using the arrays observation_count giving the number of observations for each timestep, and observation_indices containing the corresponding indices of those observations used to select the appropriate rows from observation_matrix and observation_variance. These arrays have been constructed with self.set_observations()

Parameters

- **initial_state_mean** (*array_like*) – state vector for initializing Kalman filter.
- **initial_state_covariance** (*array_like*) – state covariance matrix for initializing Kalman filter.
- **engine** (*str, optional*) – Engine to be used to run sequential Kalman filter. Either "numba" or "numpy". The default is None, which means that the default Class setting is used.

Return type
None

`run_smoother()`

Run Kalman smoother.

Calculate smoothed state means and covariances using the Kalman smoother.

`set_matrices(transition_matrix, transition_covariance, observation_matrix, observation_variance)`

Method to set matrices of state space model.

Parameters

- **transition_matrix** (*numpy.ndarray*) – State transition matrix
- **transition_covariance** (*numpy.ndarray*) – State transition covariance matrix.
- **observation_matrix** (*numpy.ndarray*) – Observation matrix.
- **observation_variance** (*numpy.ndarray*) – Observation variance.

Return type
None.

`set_observations(oseries)`

Construct observation matrices allowing missing values.

Initialize sequential processing of the Kalman filter.

Parameters
oseries (*pandas.DataFrame*) – multiple time series

simulate(*observation_matrix*, *method='smoother'*)

Method to get simulated means and covariances.

Parameters

- **observation_matrix** (*numpy.ndarray*) – Observation matrix for projecting states.
- **method** (*str, optional*) – If "filter", use Kalman filter to obtain estimates. If "smoother", use Kalman smoother. The default is "smoother".

Returns

- **simulated_means** (*list*) – List of simulated means for each time step.
- **simulated_variances** (*list*) – List of simulated variances for each time step. Variances are diagonal elements of simulated covariance matrix.

metran.kalmanfilter.**filter_predict**(*filtered_state_mean*, *filtered_state_covariance*, *transition_matrix*, *transition_covariance*)

Predict state with a Kalman Filter using sequential processing.

Parameters

- **filtered_state_mean** (*numpy.ndarray*) – Mean of state at time t-1 given observations from times [0…t-1]
- **filtered_state_covariance** (*numpy.ndarray*) – Covariance of state at time t-1 given observations from times [0…t-1]

Returns

- **predicted_state_mean** (*numpy.ndarray*) – Mean of state at time t given observations from times [0…t-1]
- **predicted_state_covariance** (*numpy.ndarray*) – Covariance of state at time t given observations from times [0…t-1]

metran.kalmanfilter.**filter_update**(*observations*, *observation_matrix*, *observation_variance*, *observation_indices*, *observation_count*, *state_mean*, *state_covariance*)

Update predicted state with Kalman Filter using sequential processing.

Parameters

- **observations** (*numpy.ndarray*) – Observations for sequential processing of Kalman filter.
- **observation_matrix** (*numpy.ndarray*) – observation matrix to project state.
- **observation_variance** (*numpy.ndarray*) – observation variances
- **observation_indices** (*numpy.ndarray*) – used to compress observations, observation_matrix, and observation_variance skipping missing values.
- **observation_count** (*numpy.ndarray*) – number of observed time series for each timestep determining the number of elements to be read in observation_indices.
- **state_mean** (*numpy.ndarray*) – mean of state at time t given observations from times [0…t-1]
- **state_covariance** (*numpy.ndarray*) – covariance of state at time t given observations from times [0…t-1]

Returns

- **state_mean** (*[n_dim_state] array*) – Mean of state at time t given observations from times [0…t], i.e. updated state mean
- **state_covariance** (*[n_dim_state, n_dim_state] array*) – Covariance of state at time t given observations from times [0…t], i.e. updated state covariance
- **sigma** (*float*) – Weighted squared innovations.
- **detf** (*float*) – Log of determinant of innovation variances matrix.

metran.kalmanfilter.**kalmansmoother**(*filtered_state_means*, *filtered_state_covariances*, *predicted_state_means*, *predicted_state_covariances*, *transition_matrix*)

Method to run the Kalman smoother.

Estimate the hidden state at time for each time step given all observations.

Parameters

- **filtered_state_means** (*array_like*) – *filtered_state_means[t]* = mean state estimate for time t given observations from times [0…t].
- **filtered_state_covariances** (*array_like*) – *filtered_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t].
- **predicted_state_means** (*array_like*) – *predicted_state_means[t]* = mean state estimate for time t given observations from times [0…t-1].
- **predicted_state_covariances** (*array_like*) – *predicted_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t-1].
- **transition_matrix** (*numpy.ndarray*) – State transition matrix from time t-1 to t.

Returns

- **smoothed_state_means** (*numpy.ndarray*) – Mean of hidden state distributions for times [0…n_timesteps-1] given all observations
- **smoothed_state_covariances** (*numpy.ndarray*) – Covariance matrix of hidden state distributions for times [0…n_timesteps-1] given all observations

metran.kalmanfilter.**seqkalmanfilter**(*observations*, *transition_matrix*, *transition_covariance*, *observation_matrix*, *observation_variance*, *observation_indices*, *observation_count*, *filtered_state_mean*, *filtered_state_covariance*)

Method to run sequential Kalman filter optimized for use with numba.

This method requires numba to be installed. With numba, this method is much faster than seqkalmanfilter_np. However, without numba, it is extremely slow and seqkalmanfilter_np should be used.

Parameters

- **observations** (*numpy.ndarray*) – Observations for sequential processing of Kalman filter.
- **transition_matrix** (*numpy.ndarray*) – State transition matrix from time t-1 to t.
- **transition_covariance** (*numpy.ndarray*) – State transition covariance matrix from time t-1 to t.
- **observation_matrix** (*numpy.ndarray*) – Observation matrix to project state.
- **observation_variance** (*numpy.ndarray*) – Observation variances
- **observation_indices** (*numpy.ndarray*) – Used to compress observations, observation_matrix, and observation_variance skipping missing values.

- **observation_count** (*numpy.ndarray*) – Number of observed time series for each timestep determining the number of elements to be read in observation_indices.
- **filtered_state_mean** (*numpy.ndarray*) – Initial state mean
- **filtered_state_covariance** (*numpy.ndarray*) – Initial state covariance

Returns

- **sigmas** (*numpy.ndarray*) – Weighted squared innovations.
- **detfs** (*numpy.ndarray*) – Log of determinant of innovation variances matrix.
- **sigmacount** (*int*) – Number of elements in sigmas en detfs with calculated values.
- **filtered_state_means** (*numpy.ndarray*) – *filtered_state_means[t]* = mean state estimate for time t given observations from times [0…t].
- **filtered_state_covariances** (*numpy.ndarray*) – *filtered_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t].
- **predicted_state_means** (*numpy.ndarray*) – *predicted_state_means[t]* = mean state estimate for time t given observations from times [0…t-1].
- **predicted_state_covariances** (*numpy.ndarray*) – *predicted_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t-1].

metran.kalmanfilter.**seqkalmanfilter_np**(*observations*, *transition_matrix*, *transition_covariance*, *observation_matrix*, *observation_variance*, *observation_indices*, *observation_count*, *filtered_state_mean*, *filtered_state_covariance*)

Method to run sequential Kalman filter optimized for use with numpy.

This method is suggested if numba is not installed. It is, however, much slower than seqkalmanfilter combined with numba.

Parameters

- **observations** (*numpy.ndarray*) – Observations for sequential processing of Kalman filter.
- **transition_matrix** (*numpy.ndarray*) – State transition matrix from time t-1 to t.
- **transition_covariance** (*numpy.ndarray*) – State transition covariance matrix from time t-1 to t.
- **observation_matrix** (*numpy.ndarray*) – Observation matrix to project state.
- **observation_variance** (*numpy.ndarray*) – Observation variances
- **observation_indices** (*numpy.ndarray*) – Used to compress observations, observation_matrix, and observation_variance skipping missing values.
- **observation_count** (*numpy.ndarray*) – Number of observed time series for each timestep determining the number of elements to be read in observation_indices.
- **filtered_state_mean** (*numpy.ndarray*) – Initial state mean
- **filtered_state_covariance** (*numpy.ndarray*) – Initial state covariance

Returns

- **sigmas** (*list*) – Weighted squared innovations.
- **detfs** (*list*) – Log values of determinant of innovation variances matrix.

- **filtered_state_means** (*list*) – *filtered_state_means[t]* = mean state estimate for time t given observations from times [0…t].
- **filtered_state_covariances** (*list*) – *filtered_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t].
- **predicted_state_means** (*list*) – *predicted_state_means[t]* = mean state estimate for time t given observations from times [0…t-1].
- **predicted_state_covariances** (*list*) – *predicted_state_covariances[t]* = covariance of state estimate for time t given observations from times [0…t-1].

4.4 Solvers

This module contains the solver that is available for Pastas Metran.

All solvers inherit from the BaseSolver class, which contains methods to obtain the object function value and numerical approximation of the parameter covariance matrix.

To solve a model the following syntax can be used:

```
>>> mt.solve(solver=ps.LmfitSolve)
```

class metran.solver.**BaseSolver**(*mt*, ***kwargs*)

All solver instances inherit from the BaseSolver class.

mt

Type
Metran instance

objfunction(*p*, *callback*)

Method to get objective function used by solver.

Parameters

- **p** (*type required for callback function*) – Parameters to be coverted by callback function into proper type and format.
- **callback** (*ufunc*) – Function that is called after each iteration. The parameters are provided to the func, e.g. "callback(parameters)".

Returns
obj – Objective function value.

Return type
float

class metran.solver.**LmfitSolve**(*mt*, ***kwargs*)

Class for solving the model using the LmFit solver [LM].

Lmfit is basically a wrapper around the scipy solvers, adding some functionality for boundary conditions.

Parameters

- **mt** (*Metran instance*) –
- ****kwargs** (*dict, optional*) – All keyword arguments will be passed onto minimization method from the solver.

Examples

```
>>> mt.solve(solver=ps.LmfitSolve)
```

References

solve(*method='lbfgsb'*, ***kwargs*)

Method to run solver and optimize parameters.

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. The default is "lbfgsb".
- ****kwargs** (*dict, optional*) – All keyword arguments will be passed onto minimization method from the solver.

Returns

- **success** (*boolean*) – True if optimization routine terminated successfully.
- **params** (*lmfit.Parameters instance*) – Ordered dictionary of Parameter objects.

class metran.solver.**ScipySolve**(*mt*, ***kwargs*)

Solver based on Scipy's least_squares method [scipy_ref].

This class is the default solver class in the Metran solve method.

Parameters

- **mt** (*Metran instance*) –
- ****kwargs** (*dict, optional*) – All keyword arguments will be passed onto minimization method from the solver.

Examples

```
>>> mt.solve(solver=ps.ScipySolve)
```

References

solve(*method='l-bfgs-b'*, ***kwargs*)

Method to run solver and optimize parameters.

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. The default is "l-bfgs-b".
- ****kwargs** (*dict, optional*) – All keyword arguments will be passed onto minimization method from the solver.

Returns

- **success** (*boolean*) – True if optimization routine terminated successfully.
- **params** (*lmfit.Parameters instance*) – Ordered dictionary of Parameter objects.

4.5 Plots

This module contains the Plot helper class for Metran.

class metran.plots.**MetranPlot**(*mt*)

Plots available directly from the Metran Class.

decomposition(*name*, *tmin=None*, *tmax=None*, *ax=None*, *split=False*, *adjust_height=True*, ***kwargs*)

Plot decomposition into specific and common dynamic components.

Parameters

- **name** (*str*) – name of oseries
- **tmin** (*str or pd.Timestamp, optional*) – start time, by default None
- **tmax** (*str or pd.Timestamp, optional*) – end time, by default None
- **ax** (*matplotlib.pyplot.Axis*) – axes to plot decomposition on
- **split** (*bool, optional*) – plot specific and common dynamic factors on different axes, only if ax is None
- **adjust_height** (*bool, optional*) – scale y-limits of axes relative to one another, by default True, only used when ax is None and split=True

Returns
axes – list of axes handles

Return type
list of matplotlib.pyplot.Axes

decompositions(*tmin=None*, *tmax=None*, ***kwargs*)

Plot all decompositions into specific and common dynamic components.

Parameters

- **name** (*str*) – name of oseries
- **tmin** (*str or pd.Timestamp, optional*) – start time, by default None
- **tmax** (*str or pd.Timestamp, optional*) – end time, by default None

Returns
axes – list of axes handles

Return type
list of matplotlib.pyplot.Axes

scree_plot()

Draw a scree plot of the eigenvalues.

Returns
ax – plot axis handle

Return type
matplotlib.pyplot.Axes

simulation(*name*, *alpha=0.05*, *tmin=None*, *tmax=None*, *ax=None*)

Plot simulation for single oseries.

Parameters

- **name** (*str*) – name of the oseries

- **alpha** (*float, optional*) – confidence interval statistic, by default 0.05 (95% confidence interval), if None no confidence interval is shown.
- **tmin** (*str or pd.Timestamp, optional*) – start time, by default None
- **tmax** (*str or pd.Timestamp, optional*) – end time, by default None
- **ax** (*matplotlib.pyplot.Axis*) – axes to plot simulation on, if None (default) create a new figure

Returns
ax – plot axis handle

Return type
matplotlib.pyplot.Axes

simulations(*alpha=0.05*, *tmin=None*, *tmax=None*)

Plot simulations for all oseries.

Parameters

- **name** (*str*) – name of the oseries
- **alpha** (*float, optional*) – confidence interval statistic, by default 0.05 (95% confidence interval), if None no confidence interval is shown.
- **tmin** (*str or pd.Timestamp, optional*) – start time, by default None
- **tmax** (*str or pd.Timestamp, optional*) – end time, by default None
- **ax** (*matplotlib.pyplot.Axis*) – axes to plot simulation on, if None (default) create a new figure

Returns
axes – list of axes handles

Return type
list of matplotlib.pyplot.Axes

state_means(*tmin=None*, *tmax=None*, *adjust_height=True*)

Plot all specific and common smoothed state means.

Parameters

- **tmin** (*str or pd.Timestamp, optional*) – start time, by default None
- **tmax** (*str or pd.Timestamp, optional*) – end time, by default None
- **adjust_height** (*bool, optional*) – scale y-axis of plots relative to one another, by default True

Returns
axes – list of axes handles

Return type
list of matplotlib.pyplot.Axes

CHAPTER

FIVE

INDICES AND TABLES

- genindex

BIBLIOGRAPHY

[LM] https://github.com/lmfit/lmfit-py/

[scipy_ref] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html

PYTHON MODULE INDEX

m

INDEX

Symbols

B

D

F

G

I

K

L

M

N

O

R

S

T

U